UiT The Arctic University of Norway

Faculty of Science and Technology

Department of Computer Science

ZeroComm: Decentralized, Secure and Trustful Group Communication

Yasiru Rathsara Witharanage

INF-3990 Master’s thesis in Computer Science - May 2023

Abstract

In the context of computer networks, decentralization is a network architecture that distributes both workload and control of a system among a set of coequal participants. Applications based on such networks enhance trust involved in communication by eliminating the external authorities with self-interests, including governments and tech companies. The decentralized model delegates the ownership of data to individual users and thus mitigates undesirable behaviours such as harvesting personal information by external organizations. Consequently, decentralization has been adopted as the key feature in the next generation of the Internet model which is known as Web 3.0. DIDComm is a set of abstract protocols which enables secure messaging with decentralization and thus serves for the realization of Web 3.0 networks. It standardizes and transforms existing network applications to enforce secure, trustful and decentralized communication. Prior work on DIDComm has only been restricted to pair-wise communication and hence it necessitates a feasible strategy for adapting the Web 3.0 concepts in group-oriented networks.

Inspired by the demand for a group communication model in Web 3.0, this study presents ZeroComm which preserves decentralization, security and trust throughout the fundamental operations of a group such as messaging and membership management. ZeroComm is built atop the publisher-subscriber pattern which serves as a messaging architecture for enabling communication among multiple members based on the subjects of their interests. This is realized in our implementation through ZeroMQ, a low-level network library that facilitates the construction of advanced and distributed messaging patterns. The proposed solution leverages DIDComm protocols to deliver safe communication among group members at the expense of performance and efficiency. ZeroComm offers two different modes of group communication based on the organization of relationships among members with a compromise between performance and security. Our quantitative analysis shows that the proposed model performs efficiently for the messaging operation whereas joining a group is a relatively exhaustive procedure due to the establishment of secure and decentralized relationships among members. ZeroComm primarily serves as a low-level messaging framework but can be extended with advanced features such as message ordering, crash recovery of members and secure routing of messages.

Acknowledgements

I would like to thank my main supervisor Associate Professor Mariusz Nowostawski for guiding me throughout this entire thesis both as a supervisor and a mentor. I immensely admire you as a role model and am grateful for all the insightful discussions on becoming a better academic researcher.

Similarly, I appreciate the contribution of my internal supervisor Professor Håvard Dagenborg for regularly rectifying the progress of the thesis with your invaluable proficiency.

Finally, I would like to express gratitude to my fiancée, family and friends for continuously being the pillars of strength in achieving success in my studies.

Table of Contents

List of Tables

List of Figures

List of Acronyms

BLOB Binary Large Object

DID Decentralized Identifier

DIF Decentralized Identity Foundation

DLT Distributed Ledger Technology

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IP Internet Protocol

MAC Message Authentication Code

MQ Message Queue

OSI Open Systems Interconnection

PKI Public Key Infrastructure

RFC Request for Comments

TCP Transmission Control Protocol

TLS Transport Layer Security

ToIP Trust over IP

URL Uniform Resource Locator

UUID Universally Unique Identifier

ZMTP ZeroMQ Message Transport Protocol

1 Introduction

The involvement of public and private organizations as centralized intermediaries in modern computer networks has given rise to a dispute where trust of communication is delivered by these external parties with self-interests [1]. Despite the security guarantees of an encrypted channel for communication, a centralized model delegates the ownership of user-sensitive data to a tech company and further extends the ability to share traced user activities with other interested parties [2]. Thus, existing communication frameworks fail to provide absolute digital trust as long as they are composed of centralized entities.

The research community is working to eliminate these centralized components involved in communication, thus resulting in more decentralized network systems. As a result of this transposition of fundamental elements, the ecosystem of the Internet has been remodelled as Trust over IP (ToIP) infrastructure by including the missing trust factor in the existing network stack [3]. The ToIP framework introduces a novel set of protocols known as DIDComm, which has been designed such that the existing messaging models and transport protocols can be easily integrated into the new ecosystem while enhancing digital trust of communication.

The DIDComm protocols unfold the potential of revamping the entire context of computer networks, including but not limited to messaging patterns used for transmitting data among the participants. However, all prior work on aligning existing transport protocols (eg: Bluetooth, WebSockets) with DIDComm have been primarily focused on point-to-point communication. To the best of our knowledge, no attempts have been made so far on the rest of the messaging patterns thus restricting the applicability of DIDComm in group-oriented network systems.

Publisher-subscriber is a widely used messaging pattern that resolves group communication effectively. Regardless of the efficiency in data transmission, security in messages has not been entailed as a core feature of this pattern and it is currently fulfilled explicitly by the corresponding applications. This unravels a possible research area to explore the possibility of combining DIDComm with pub-sub pattern, thus embodying trust of communication into the pattern itself independent of the application using it.

In addition to the trust concerns, pub-sub implementations tend to be more centralized with the use of dedicated message-queue brokers to hold the messages being transmitted, thus conflicting with DIDComm's core properties. However, ZeroMQ[1] serves as a library with low-level network sockets which can be combined to construct high-level messaging patterns such as pub-sub while eliminating centralized message-queue brokers [4]. When combined with DIDComm protocols, ZeroMQ yields the potential of a truly decentralized secure pub-sub system and a long list of use-cases waiting to be unfolded together with this powerful combination.

1.1 Problem Definition

The transport-agnostic property of DIDComm plays a vital role in adaptability as it preserves the underlying messaging patterns and technologies in facilitating safe communication in existing software systems. A number of implementations have had success using transport protocols such as HTTPS, WebSockets and Bluetooth while a considerable amount of research work is invested in novel transport mechanisms [5].

Prior work on the transport layer of DIDComm is focused on the conventional client-server pattern, thus limiting the growth of decentralized applications with group-oriented networks where trustful communication is deemed crucial. For instance, the publisher-subscriber can be regarded as a widely-used messaging pattern that still remains to be explored with DIDComm.

Our thesis demonstrates that a trustful and decentralized group-communication model can be constructed by combining ZeroMQ and DIDComm protocols. Further, we investigate the following problems in the course of achieving this objective.

1. How does trust in modern technological communication impact publisher-subscriber pattern and how does it differ from bi-party models?
2. How do decentralization, trust and group communication inter-operate with each other?
    (a) To what degree can contradictory factors of each element be preserved for the delivery of a trustful decentralized group communication model?
    (b) How will these overlapping design choices result in alternative solutions with different trade-offs?

[1]Also referred to as zmq or ØMQ (more information is available at https://zguide.zeromq.org)

3. How can the requirements of a trustful decentralized group be transformed into a technically feasible strategy? What communication protocols of trust should be considered in the design of a solution?
4. What mechanisms should be formulated for evaluating such a model? How should experiments be designed to involve an adequate set of metrics for evaluation?

1.2 Methodology

In the attempt of formulating a curriculum for the discipline of computing, Peter J. Denning et al. have defined the core aspects which can generally be applied to any study involved with the field [6]. This specification formally branches the discipline into three major paradigms with the intention of distinguishing areas of competence but the processes may be intertwined with each other in practice.

- *Theory*: Rooted in mathematics and provides the ability to explain in terms of definitions and prove relationships or theorems associated with objects
- *Abstraction*: Scientific approach of the study which iterates through models, predictions, experiments and analysis of data for the validation of a hypothesis
- *Design*: Engineering-based procedure with a repetition of requirements, specifications, design, implementation and testing of a system

This study predominantly adheres to the *design* paradigm as we primarily focus on constructing a novel solution. In particular, our proposed model is expected to deliver a functional system by combining desirable properties of different frameworks. Hence, the workflow of our thesis conforms to an iterative procedure of; literature review, identification of system requirements, constraints and trade-offs, proposition of a solution, system design with a composition of components and implementation followed by a testing process of the proposed solution. In addition, the study also comprises an evaluation of the system with a set of befitting metrics and an analysis based on properties and design choices, thus partly including the elements of a scientific approach as well.

1.3 Context

ZeroComm can be regarded as a project which spans numerous dominant contexts related to the discipline of computing such as computer security, networking, distributed systems and software architecture. In terms of security, ToIP components have been gaining traction with the recognition and standardization by W3C on account of its ability to provide decentralized solutions in a trustworthy manner [7, 8]. Several applications have already been deployed and currently are in use, mostly as digital wallets for managing verifiable credentials[2]. This study serves as a lower-layer component of the overall ToIP ecosystem specifically in communication layer with secure transmission of messages and it is expected to conform with DIF[3] (a governing organization of decentralized identity) along with the rest of DIDComm community, including Hyperledger[4].

1.4 Goals and Contribution

As the major contribution of our study, we deliver a decentralized group communication model with DIDComm while encompassing the following attributes.

- Based on only queues as the core date-structure
- Message integrity with end-to-end encryption
- Privacy with distinctive keys shared by a secure handshake
- Elimination of centralized parties in all phases of the group messaging flow
- Globally accessible by any interested party
- Resilience to ad-hoc nature of the pattern (eg: dynamic joins and leaves, decoupled relationship between publishers and subscribers)

Further, comprehensive analysis along with an evaluation is also expected to provide a guidance for any further development related to the specific context as well as assist in standardizing group messaging protocols in terms of DIDComm.

[2]Digital wallet examples: https://docs.trinsic.id, https://www.evernym.com/solution
[3]Official website: https://identity.foundation
[4]More information is available at https://www.hyperledger.org/about

1.5 Outline

The thesis commences with exploring the fundamentals of ToIP, DIDComm and ZeroMQ internals as initial guidance of the basic components in our study. Chapter 3 describes the minimal set of DIDComm protocols required for constructing a secure group-oriented communication model including their procedures and corresponding message structures. Chapter 4 discusses design goals and possible alternatives, which is then followed by the interaction of domain-related components, organization of message-queues, propagation of status within a group and formation of members along with overlay networks. Chapter 5 dives into implementation-specific details of our solution such as the system architecture, functions of a group member, types of messages involved in the group protocol and external dependencies. Chapter 6 includes the results of conducted experiments to evaluate our proposed model preceded by a definition of an appropriate set of metrics for a proper analysis of ZeroComm. Chapter 7 provides an extensive discussion by reflecting on internals, core features and design choices of the solution while section 8 concludes the study with significant remarks and future work.

2 Background

2.1 Trust Over IP

Since the late 1960's, the fundamental concept of connecting and passing data between two computers had been researched until it evolved into the current matured state of the Internet, which is a highly advanced, diversified and complex network with billions of nodes. Due to the fact that this evolution commenced with the simple objective of inter-machine communication and continued with immense growth driven by ceaseless requirements, the underlying concepts of this technology associate more with machines rather than with people. In simple terms, the Internet has evolved without an identity layer incorporated into its core for an ideal representation of the participants in communication.

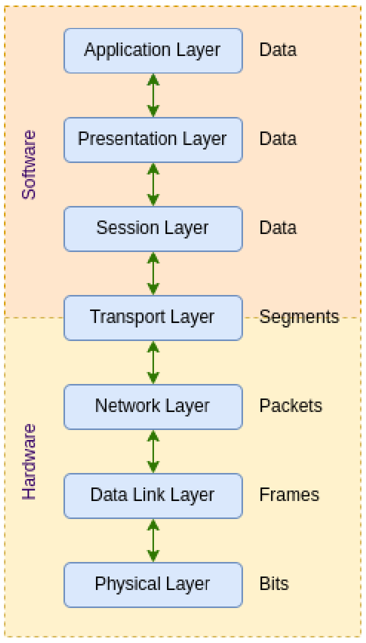


Figure 1: OSI network model

This contradictory dilemma has led to a number of workarounds for integrating user-related features into existing software systems, mostly in the application layer of the OSI model (Figure 1). In addition to the burden of reforming user-related features with additional effort and complexity, these solutions fail to provide an ideal setup where identity should technically operate in the lower layers of the network stack and administration of an identity should be only within its owner's domain. The current application-based identity solutions lead to even more unethical consequences such as surveillance, data being shared with external parties, data breaches and exposure of PII (Personal Identifiable Information), while several studies have been conducted to assess this risk statistically [9]. The continuation of this architecture by evading its core vulnerabilities and attempting service-level provisional fixes will only lead to an escalation of thefts and deceptions while further eroding trust in the Internet. In fact, this culprit has been addressed by the founders of the Internet despite they could not foresee it in advance [10].

Modern network systems can be highly 'secure' with the adaption of encryption and user-based security mechanisms but can not be considered 'safe' as long as privacy concerns are not addressed sufficiently. It is also worthwhile to note that the former does not necessarily lead to the latter. This situation becomes even more complicated with the introduction of IoT devices and non-human entities in need of a web identity.

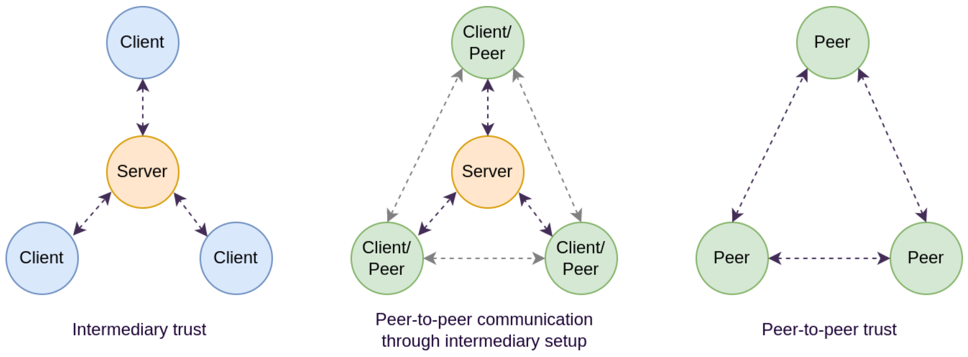


Figure 2: Existing trust models

The necessity of identification boils down to the broader subject of 'trust', since identity is mandatory to establish trust between participants. As an initiative, the fundamental unit of a new trust framework is constructed based on a peer model as analogous to the real-world (Figure 2), representing both direct and transitive relationships (as an example of the latter, trust can be established between A-C, if trusts between A-B and B-C already exist in the given context). This eliminates the involvement of a 3rd party for identity management and delegates the entire authority of identity only to its owner.

However, technology alone is insufficient for constructing a trust model since a significant portion of the trust involves a psychological factor. Hence, the peer-to-peer model was extended further by redefining the network stack with respect to human involvement and thus resulting

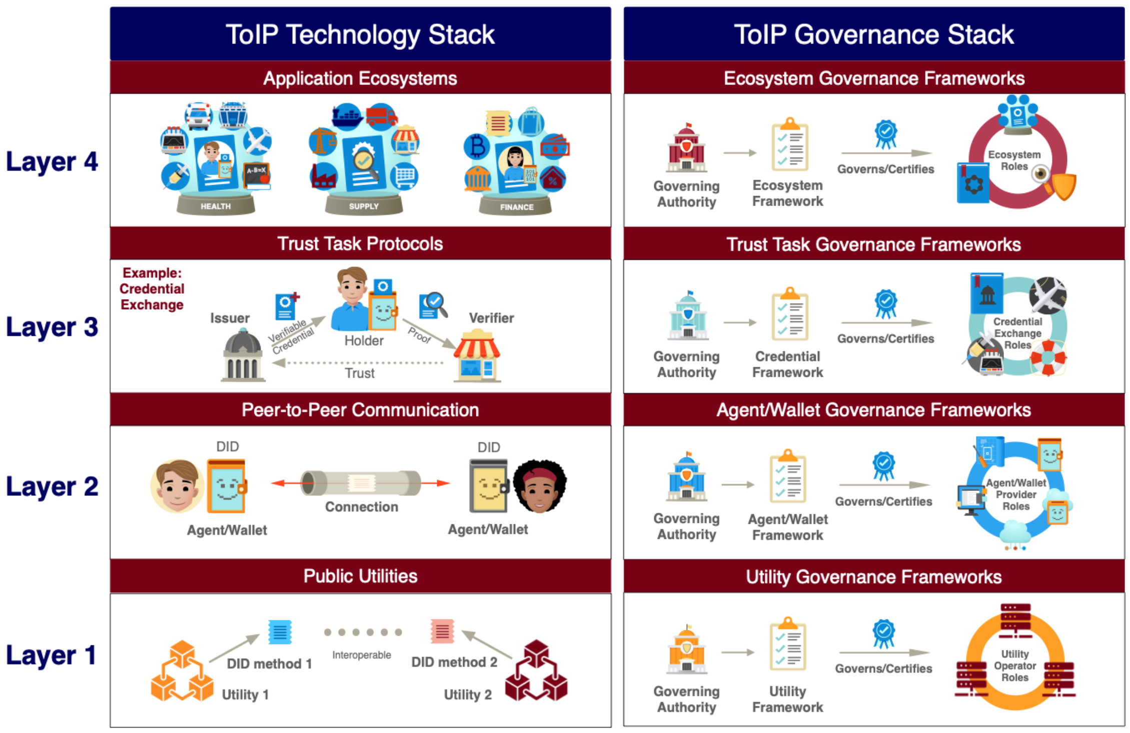


Figure 3: Two-sided and four-layer ToIP stack [3]

in two distinctive but associative sets of layers. As shown in Figure 3, a governance stack is introduced along with its technological correspondence. This includes governance authorities and frameworks formed to define business, legal and social terms which are applicable for each layer and exist in either formal or informal, computer code or legal document. This combination of technology and governance is expected to maintain adequate trust levels while mitigating the pitfalls of the existing the Internet model.

In essence, the ToIP framework relies on 3 fundamental components, namely, Decentralized Identifiers (DIDs), DIDComm and Distributed Ledger Technology (DLT). DIDs are used for the unique recognition as well as representation of participants while DIDComm serves as a secure communication channel constructed atop individual DIDs. DLT provides a single source of truth that is globally accessible by all the parties and also resolvable through DIDs in seeking any additional information.

Layer 1 in the ToIP stack is associated with foundational entities such as DID methods, Verifiable Data Registries (VDR), Public Key Infrastructure (PKI) while layer 2 sets up the communication links between individual entities via agents, digital wallets and DIDComm. Establishing trust relationships between participants based on such constructed links is accomplished by layer 3 whereas the top-most layer delivers market applications with healthy trust ecosystems built upon lower layers of the framework. In addition, authorities and frameworks which operate at each layer are authorized and supported by their governing complements. As briefly described here, all layers utilize elements of the lower ones to support the layers above. Hence each technical component in every layer, including but not limited to DIDComm, plays a vital role in fulfilling the ultimate objective of an ideal network framework embedded with trust.

2.2 Decentralized Identifiers

For the realization of a trust framework as described in the previous section, each node in a network should possess and be represented by a specific form of identification, which is defined as a Decentralized Identifier (DID) in the ToIP context. In order to alleviate fraudulent behaviour within a system, this identity should not be capable of modification (immutable) but only creation and revocation. Further, it should be persisted in a globally accessible storage without being impacted by any centralized authority, such that it satisfies the trust requirement of ToIP. To this end, DLTs are preferred and heavily adopted for this specific use-case despite distributed databases and certain web-based solutions can generally be utilized as well.

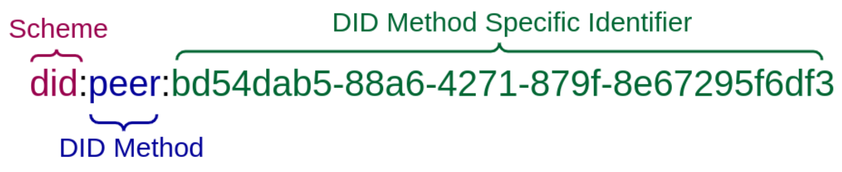

Scheme
DID Method Specific Identifier
did:peer:bd54dab5-88a6-4271-879f-8e67295f6df3
DID Method

Figure 4: A sample DID with components

Due to the availability of a vast number of storage alternatives, these identifiers do not entail a precise structure but rather a general form with constraints to be recognizable as a DID and moderate enough to be distinguishable with respect to the storage mechanism. Figure 4 depicts the basic components of a DID. Note that, the scheme is a section with the fixed value ‘did’ while DID method resembles the underlying storage protocol (eg: *key*, *ethr*, *btcr*, *github*). However, to accommodate private relationships while being ledger-agnostic, it is also possible to use DIDs without any specific external persistence (eg: *peer*). The rightmost segment is unique to the corresponding DID method and its syntax can vary accordingly.

The identifier resolves to a document known as DID Document (DIDDoc) which exists in a storage based on the DID method being used and describes a particular subject such as a person, an organization, an IoT device or a pet. In the case of a peer DID, the construction of this document should follow a specific procedure such that it is self-contained within the DID and can be parsed by any subsequent recipient. The subject of a document does not necessarily need to be the owner of the corresponding DID (eg: DID of a pet can be owned by a person).

DID resolution performs a similar service as Domain Name System (DNS) resolution but in contrast, the former does not follow a concrete implementation due to the possible interference of multiple ledger protocols. Hence it should rather be considered as an abstract function by the application layer. DID is feature-wise more similar to a Uniform Resource Name (URN) and its format can further be extended to function as a URL with query parameters and fragments. These extended DID URLs are particularly useful in directing a recipient towards a specific resource component, such as the exact public key to be used from multiple keys in order to communicate with a peer.

DID Document does not impose any constraint on the content but often contains a public key for the decryption of messages sent by the subject (or encryption of messages intended for the subject) along with one or multiple service endpoints for further communication. However, it is not recommended to include any PII (Personal Identifiable Information) since DID Document is persisted and exposed to the public, mostly via distributed ledgers. Nevertheless, this implies that DID does not simply serve as a form of identification but also serves as a commencement of a secure communication channel via the introduction of keys and service endpoints. Despite the similarities with real-world identifiers, DID should be treated with respect to the context of a relationship, as it is generally not recommended to use the same DID across multiple relationships (unless it is specifically viable). This reduces the traceability of a particular subject beyond a single relationship together with the vulnerability of exposing all messages associated with the DID.

One of the key characteristics of the ToIP framework is interoperability, which implies that each service component including these identifiers should function irrespective of the underlying technology. This is similar to how the Internet evolved and enabled a wide variety of technologies to communicate with each other by means of a single global network. Likewise, interoperability necessitates the trust framework to standardize particular fields of a DID Document as demonstrated in Figure 5, such that they are syntactically well-understood by different applications once resolved.

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ],
  "id": "did:example:123456789abcdefghi",
  "authentication": [{
    "id": "did:example:123456789abcdefghi#keys-1",
    "type": "Ed25519VerificationKey2020",
    "controller": "did:example:123456789abcdefghi",
    "publicKeyMultibase": "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
  }]
}
```

Figure 5: A sample DID Document

2.3 DIDComm

DIDComm is a set of protocols defined within the ToIP framework to enable safe communication among the participants. However, the transport layer which is essential for the transmission of individual messages functions only as an abstract component of these protocols, thus leading DIDComm to be **transport-agnostic** and utilized with the existing network infrastructure by the systems. Therefore, it can be constructed atop any transport protocol such as HTTPS, Bluetooth, WebSockets and Near Field Communication (NFC). In essence, DIDComm does not operate as an entirely new technology but only standardizes and aggregates existing components to formulate a safe messaging framework.

DIDComm is designed to be asynchronous, simplex and based on peers since this simple model is convenient to provide **interoperability** across different use-cases. Specifically, these naive design choices seem to be more applicable than the conventional client-server model, for a world growing with ubiquitous and transient communication devices. This simple model does not merely employ DIDComm in cases that had not been addressed adequately before, but also provides extensibility to construct even the traditional patterns securely if needed, such as the client-server model.

In the networking context, DIDComm's functionality is equivalent to how TLS protocols enable secure transmission of data throughout the web. However, it eliminates the centralized intermediaries such as Certificate Authorities (CAs), thus resulting in a more decentralized peer-to-peer communication model and displaces the associated pitfalls of CAs such as centralization with the vulnerability of attacks, single point of failure, additional costs and scalability limitations [11]. Basically, DIDComm does not only provide security in communication but also trust and self-sovereignty which are equally managed by the participants.

Each agent involved with communicating via DIDComm is represented by a unique DID. As described in the previous section, DID resolves to a DID Document which at least contains the subject's public key and an endpoint. This information can be retrieved by any interested party for setting up a secure communication channel while encrypting messages using the resolved public key. Therefore, DID functions as an initiator of communication in this case despite that it primarily serves as an identifier in general. Further, this functionality enables a wide range of application-level use cases such as credential exchange, secure messaging and authentication. In fact, these interactions are depicted in Figure 3 where DIDComm (Layer 2) utilizes the underlying DIDs and storage utilities (Layer 1) to deliver the corresponding application services (Layer 3).

The agnosticism of DIDComm preserves any fundamental design pattern of communication used along with the transport. For example, a system with producer-consumer pattern based on HTTP can impose DIDComm conventions to enable safe communication without having to alter the underlying network. This can further be beneficial in cases where **decentralization** and **safety** are highly concerned factors but could not yet be accomplished due to the inherent nature of conventional network models.

DIDComm operates in the upper layers of the OSI model while utilizing the transport layer, but it embeds security not at the transport level but in messages themselves. This is due to its security guarantees being derived independently from the underlying transport. This particular attribute results in the possibility of constructing even higher-level conceptual paradigms such as multiplexing of different transport protocols at the expense of a single initial handshake. Thus, DIDComm provides the capability of extending globalized communication even further in the Web 3.0 context.

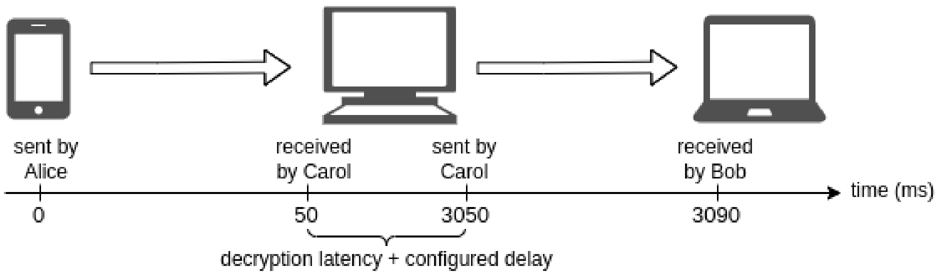


Figure 6: Delay mechanism of mediators (time axis is not to scale)

Each message of communication in modern times traverses through multiple intermediary nodes (*mediators*) until it is received by the intended recipient. This imposes a risk of tracking data maliciously even though it is mitigated by solid guarantees of encryption in TLS protocols. However, DIDComm requires the message to be encrypted in a recursive manner such that each mediator will decrypt the outermost layer to fetch the succeeding node of the route and forward the residual BLOB accordingly. Besides the fact that core content of the message is obscured from the mediators, this also hinders the discovery of other nodes in the link for a given specific route, including the sender and receiver. This **secure routing** of DIDComm is further extended by including intermediate delay mechanisms to obfuscate the temporal patterns of communication. For example, a message can be configured to be sent by a particular mediator with a random delay of 1-10 seconds (Figure 6). Moreover, the mediators are capable of enforcing *rewrapping* for a given message, in which the decrypted residual will be extended with an additional layer of encryption by the mediator. This will enable dynamic alterations of the message route along with security enhancements to the message content.

2.4 ZeroMQ

2.4.1 Message-Queue model

The paradigm of client-server architecture has dominated web-based applications due to its simplicity and applicability over a wide range of use-cases. However, this pattern fails to address a number of concerns particularly with distributed systems while limiting its scalability. One of the known issues can be listed as the latency cost of connection establishment in HTTPS where an initial handshake is accomplished between the involved client and server [12]. Despite the improvements being made to the protocol (TLS 1.3 reduces the cost of this handshake in terms of round trip times [13]), this synchronous behaviour can lead to extensive utilization of network and hardware resources, specifically where multiple sessions are established among a set of participants. At the system level, using a client-server model may complicate the entire architecture in terms of relationships, such as in a distributed system with numerous micro-services in need of interacting with each other. In general, the conventional model is not the one-size-fits-all solution for communication use-cases despite its convenience.

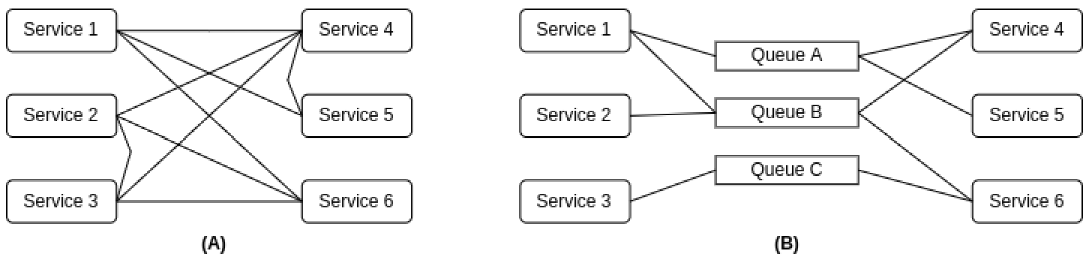


Figure 7: Client-server (A) vs message-queue architecture (B)

As an alternative, Message Queue (MQ) model offers a different perspective on communication patterns, in which systems are based on FIFO data structures and interaction among components is achieved through messages instead of direct one-to-one relationships (Figure 7). Designing an MQ system may comparatively require more effort and comprehension, but it often results in less complicated and more scalable system architectures. Further, it enables a set of noteworthy rich design patterns such as publisher-subscriber, producer-consumer and master-worker, which essentially exploits message queues as the core model. MQ model has evolved to deliver a wide variety of products with their own technical, transport and architectural differences but nevertheless, this study uses only ZeroMQ as the underlying transport protocol.

2.4.2 Properties

Generally, MQ systems comprise a centralized component (eg: Kafka [14], RabbitMQ[5]), which is commonly known as a *broker* to manage queues and their corresponding relationships with peers. In contrast, ZeroMQ eliminates the prerequisite of a broker and offers the capability of a more **decentralized** peer-to-peer system by storing data queues within the nodes themselves. Data transferred via ZeroMQ is structured as **frames** (Figure 8) to support discrete messages

[5]More information is available at https://www.rabbitmq.com

but metadata in each frame can be combined to resemble a *multi-part* message. Transmission of these frames is carried out adhering to a low-level specification known as ZeroMQ Message Transport Protocol (ZMTP).

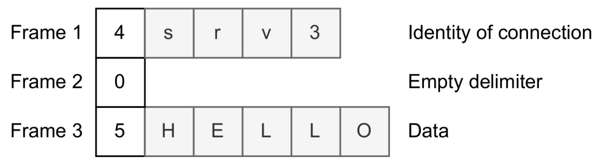


Figure 8: Frames of a multi-part message (each frame comprises size and content)

At the wire level, ZeroMQ does not constrain the necessity of a particular data format or encoding mechanism, thus allowing the programmer to use **any serialization** (eg: JSON[6], Avro, Protobuf, MessagePack) as long as the messages are transformed into BLOBs. A richer API (Application Programming Interface) is available to handle application-level message structures, in addition to the basic *send* and *receive* functions which truncate messages with a configured buffer size.

Similar to DIDComm, ZeroMQ also exhibits an **agnosticism in transport** layer to a certain extent, but it is limited by the protocols which already have been defined in ZeroMQ standards such as unicast (TCP, inter-process, intra-process) and multicast (PGM[7] [15], EPGM[8]) transports. If TCP is used as the transport, individual nodes can communicate with each other via default (public or private) IP addresses whereas intra-process allows defining local addresses as required. Interactions with the transport are decoupled from the application layer as messages are sent and received **asynchronously** by an I/O thread without blocking the main flow. It is only required to use the precise socket types within the application since the underlying connection establishments and overlay networks are entirely managed by ZeroMQ. As an example, ZeroMQ allows clients to be bootstrapped even before the server (as opposed to a conventional network), establishes connections once they are reachable (thus known as **disconnected transport**) and reconnects upon any failure of the connection.

Even though ZeroMQ advocates brokerless architecture, it also provides means for integrating **intermediaries** such as proxies, queues, forwarders and brokers as required. This only de-

[6]JavaScript Object Notation (more information is available at https://www.json.org/json-en.html)
[7]Pragmatic General Multicast
[8]Encapsulated Pragmatic General Multicast (see https://linux.die.net/man/7/zmq_epgm)

mands fundamental socket types in the implementation but yields higher-level services such as the dynamic discovery of nodes, interconnection between multiple clients and services, bridging different transports and graceful shutdown of networks. In fact, ZeroMQ can be regarded as a library with low-level blocks which serves multiple ways to build enormously large but more efficient software systems.

MQ applications are generally deployed as distributed systems with servers interconnected by a specific transport. But it is worthwhile to note that, ZeroMQ can also be **used internally** within a single computer by means of threads or processes as individual nodes with *inproc* (intra-process) or *ipc* (inter-process) as transport, respectively. This paves the way to construct more efficient intra-node communication (eg: worker models) regardless of any programming language or operating system. Above all, ZeroMQ maintains **abstraction** across the spectrum of transports and it allows an implemented solution to be easily transformed into a different granularity of nodes without any code modification. This can specifically be useful in simulations or experiments of more complex patterns with a minimal cost of infrastructure.

2.4.3 Patterns

Regardless of all the remarkable features, the essence of ZeroMQ is entangled with its ability to enable high-level design patterns by the mere integration of different socket types. ZeroComm utilizes two such design patterns to construct the group communication model, which are described in the following sub-sections.

(a) Request-Reply

Although the core model of ZeroMQ is based on message queues, it also provides sockets to emulate the fundamental communication pattern, i.e. request-reply relationship between two peers. This can be achieved conveniently via the combination of built-in socket types, *REQ* and *REP*. However, it communicates only with one peer at a time even though the socket allows the application to bind or connect with multiple peers. Further, its message flow expects a synchronous behaviour thus restricting peers from sending multiple messages until the entire round-trip of a message is completed. Nevertheless, these complications can be avoided by using *ROUTER-DEALER* socket pair which basically serves as an asynchronous *REQ-REP* combination despite the minute differences.

(b) Publisher-Subscriber

From passing messages within a single micro-service to being the backbone of a large-scale distributed platform, pub-sub pattern is widely adopted by the systems in a variety of contexts. In essence, this enables a set of interested parties to process messages continuously based on their distinctive interests, while messages are being generated and sent by a set of publishers. ZeroMQ provides built-in socket types *PUB* and *SUB* to resemble the functionality of actors involved in this pattern, thus eliminating the necessity of a centralized component to manage the queues of data (eg: proxy, forwarder, broker). However, different combinations of ZeroMQ socket types can potentially be exploited to design and implement a pub-sub system enriched with higher-level features such as message-queue brokers, fault-tolerant servers, failure-recovery mechanisms, efficient transmission of messages and elimination of laggy subscribers.

3 DIDComm Protocols

This chapter provides an in-depth exploration of DIDComm protocols which are essential to construct a group communication model and how they were implemented in our solution. In particular, we start with more specific and lower-level details in section 3.1, such as packing and unpacking procedures of individual messages along with required data structures and cryptography dependencies. Section 3.2 explains how these packed messages can be used in a DIDComm handshake in order to establish secure connections between two standalone agents.

3.1 Message Packing

Implementation of a proper encryption mechanism is vital to deliver secure messages via DIDComm while preserving data integrity and confidentiality. This can be achieved in multiple ways but the protocol defined in Aries RFC-0019 is specifically chosen for this purpose mainly due to the reasons of interoperability [16]. As further clarification, Hyperledger Aries plays a leading role in the DIDComm community and hence adhering to the same encryption process will not only guarantee the desired properties but also the communication with agents by different vendors. This specification also defines two types of encryption as *anoncrypt* and *authcrypt*, where former is used when the sender's anonymity should be preserved while the latter when the sender should be revealed. Only *authcrypt* is considered for our implementation but note that packing and unpacking functions are abstracted via a generic interface such that any customized implementation can be integrated conveniently.

A prerequisite for the encryption algorithm is that the sender should be aware of a public key as well as an endpoint of the recipient to transmit the message. This information can be made available to the sender via an out-of-band invitation or a DIDDoc of the recipient. In addition, the sender must be able to access its own key pair, ideally generated for this relationship. All the public key pairs used in this algorithm are constructed using X25519 elliptic curve function with a length of 32 bytes [17].

The cryptography related functionalities such as key generation, encryption and decryption are imported from the external library *libsodium-go*, which is essentially a cgo wrapper of its native

C library implemented by extending *NaCl* core implementation [18]. This combined tech-stack of cgo introduces certain caveats such as performance overheads and complications in debugging. However, it was decided to proceed with *libsodium-go* since our implementation should adhere to the conventions of RFC-0019 and cgo library provides all the required functions as identical to the core library. In contrast, the in-built crypto package of golang uses google's *tink* library for its underlying cryptography functions.

3.1.1 Structure

```
{
    "protected": "eyJlbmMiOiJ4Y2.....05qIn19XX0=",
    "iv": "M1GneQLepxfDbios",
    "ciphertext": "iOLSKIxqn_kC.....qfvykGs6fV",
    "tag": "gL-lfmD-MnNj9Pr6TfzgLA=="
}
```

Figure 9: Authcrypt message structure

Despite the agent is solely responsible for the encoding process and messages are transmitted as BLOBs on the wire, a definite structure is adopted from RFC-0019 in order to align with the standards of the community as shown in Figure 9. This contains an encoded initial vector (a.k.a nonce), the encrypted message as ciphertext and a set of protected header values which are necessary for the decryption of the message. Additionally, it contains a *tag* which is commonly known as Message Authentication Code (MAC), to be used as a proof of authenticity and data integrity. Figure 10 shows the plaintext version of *protected* attribute and in this example, *recipients* array contains two items resembling either two different subjects or two end-devices of the same subject. The latter scenario intends to allow the message to be read from one of the recipient's multiple devices (eg: mobile phone, laptop, tablet). As a general rule of thumb in such cases, encryption should be performed for as many recipient keys as possible.

3.1.2 Process

Sender initially generates a random non-negative integer (nonce) and its base64 encoded string is included in the protected header *iv* of the recipient. A shared symmetric key is generated afterwards, which will ultimately be used for the encryption of the underlying message. This content encryption key (*cek*) is encrypted using libsodium's *crypto_box_easy* function by providing the generated nonce, receiver's public key and sender's private key as parameters. This method is used as opposed to *crypto_box* function suggested in RFC-0019 since libsodium discourages its usage and recommends *crypto_box_easy* function instead[9]. The base64 encoded string version of the encrypted key is included as an attribute of the recipient since it varies from one peer/device to another.

```
{
    "enc": "xchacha20poly1305_ietf",
    "typ": "JWM/1.0",
    "alg": "Authcrypt",
    "recipients": [
        {
            "encrypted_key": "L5XDhH15S....a6qe4JfuAz",
            "header": {
                "kid": "GJ1SzoWzavQ....V4ct3LXKL",
                "iv": "a8IminstXHi54_J-Je5IWlOcNgSwD9TB",
                "sender": "ftimwiiYRG7....qCixKJk="
            }
        },
        {
            "encrypted_key": "eAMiD6GDmO....7QNSulYWiE16Y",
            "header": {
                "kid": "HKTAiYM8c....MBxP2i1Y92zum",
                "iv": "D4tNtHd2rs65EG_A4GB-o0-9BgLxDMfH",
                "sender": "sJ7piu4UDu....ftwH8_EYGTL0Q="
            }
        }
    ]
}
```

Figure 10: Protected headers of authcrypt message

The receiver's public keys are encoded in base58 and included in the *kid* header of each recipient section of an *authcrypt* message as shown in Figure 10. These keys will be used by the recipient to further correlate with respect to the set of targeted devices or subjects, to which the message is intended. Encoding in base58 provides more human readability compared to base64, by eliminating visually identical (0, O, I, l) and alphanumeric characters (/, +). Sender's public key is encrypted using *crypto_box_seal* library function by providing the receiver's public key. In particular, this cryptography method generates an ephemeral key pair, encrypts the sender's public key using the ephemeral private key, attaches its corresponding public key to ciphertext and discards the ephemeral private key. Hence, even the sender is unable to decrypt the message once encrypted and it also preserves the sender's anonymity despite that the message in this case is the sender's public key itself.

[9]See https://libsodium.gitbook.io/doc/public-key_cryptography/authenticated_encryption#notes

This entire payload shown in Figure 10 is encoded in base64 to be attached as a string version in *protected* attribute of the final *authcrypt* message. The message to be sent is encrypted using *cek* along with a new initial vector. An AEAD (Authenticated Encryption with Associated Data) algorithm known as *ChaCha20-Poly1305* is used for this encryption which is essentially a combination of a stream cipher (ChaCha20) and an authenticator (Poly1305). Apart from encrypting the message, it demands additional data which will not be encrypted, but instead authenticated. In this case, the set of protected header values is used for additional data and the MAC generated in the encryption process is attached as *tag* of the message.

It should be noted that a different nonce is used to encrypt the message in contrast with the one used for *cek*, since it minimizes the unnecessary correlation between two distinctively encrypted ciphers. Nonce does not require it to be confidential as its primary objective is to distinguish a particular message from any duplicates. Hence, it is only included as the base64 encoded version in *iv* field of the *authcrypt* message. In our ZeroComm implementation, a noise (slice of bytes) is appended to each nonce value in order to satisfy the array length required by the underlying *cgo* function.

```
{
     "protected": "b64URLencoded({
        "enc": "xchachapoly1305_ietf",
        "typ": "JWM/1.0",
        "alg": "Authcrypt",
        "recipients": [
            {
                "encrypted_key": base64URLencode(libsodium.crypto_box(my_key, their_vk, cek, cek_iv))
                "header": {
                      "kid": "base58encode(recipient_verkey)",
                      "sender" : base64URLencode(libsodium.crypto_box_seal(their_vk, base58encode(sender_vk)),
                      "iv" : base64URLencode(cek_iv)
                }
        },
        ],
    })",
    "iv": <b64URLencode(iv)>,
    "ciphertext": b64URLencode(encrypt_detached({'@type'...}, protected_value_encoded, iv, cek),
    "tag": <b64URLencode(tag)>
}
```

Figure 11: Authcrypt message format with functions

Figure 11 manifests a summarized version of an *authcrypt* message along with corresponding encryption and encoding mechanisms for each attribute. Their counterpart methods of decoding and decryption should be used by the recipient in a more or less inverted flow of actions. As a final note, ZeroComm implementation follows an RFC defined by the Hyperledger community, which has been recently modified by DIF thus leading to version 2.0 of the protocol with several changes.

3.2 DID Exchange

The essence of DIDComm heavily relies on the individual DIDs of each peer participating in a communication channel. A DIDComm connection can be established once DIDDoc of each peer is shared with the other participant since it provides an endpoint to communicate with, a public key for encryption protocols and other additional information as required. Hence, the exchange of DIDs prior to a safe messaging session is deemed crucial for any peer intending to support DIDComm. The implementation considered in our study complies with RFC-0023 which is formulated by Hyperledger Aries in order to preserve interoperability and maintain community standards [19]. This procedure has two major roles; a party that initiates a handshake protocol (*requester*) and a party that responds to this initiation (*responder*).

3.2.1 Invitation

As it seems paradoxical to establish a DIDComm connection using DIDComm itself, the preliminary phase of exchange protocol is carried out via alternative means of communication such as email, QR code[10], SMS (Short Message Service) [20] and other similar methods. To this end, a responder may send out an invitation (at minimum with an exchange endpoint and a public key), such that a requester's agent can process the information and reply back with an encrypted connection-request to the provided endpoint. RFC-0434 provides a comprehensive description of the flow and structure of this invitation, which is known as *Out-Of-Band* protocol as the invitation is transmitted externally [21]. However, a minimal version (Figure 12) is used for ZeroComm implementation since it only serves as a prototype in our case. Further, RFC-0434 defines the roles involved as *sender* and *receiver* which correspond to *responder* and *requester* in the subsequent DID-exchange protocol, respectively.

All the identifiers in DIDComm messages are generated as UUIDs and *id* property of the invitation will be used as the parent thread ID for the rest of the messages yet to be followed in the exchange protocol, allowing them to be correlated with a single sequence. The *label* attribute

[10]Quick Response code (more information is available at https://en.wikipedia.org/wiki/QR_code)

```
{
  "@type": "https://didcomm.org/out-of-band/1.0/invitation",
  "@id": "118ccff0-9782-4a4f-85ad-2af148087e82",
  "label": "alice",
  "from": "did:peer:1z7pZmubmVGYtCNs93sZhC9LdnXkRpjUWjkn3nE4rz75j",
  "services": {
     "@type": "did-exchange",
     "@id": "165ba09c-4a64-4258-a349-be390f0aff60",
     "recipientKeys": ["u7HuQ3jixfA7e7578A....8bHOqyL3oBLKL5Qc="],
     "serviceEndpoint": "tcp://127.0.0.1:5555",
  },
}
```

Figure 12: Structure of a sample invitation

only serves as a reference of the sender to the other party, in case the receiver connects to multiple peers for communication. Despite the abundance of various DID methods (eg: did:btcr, did:ion, did:git), ledger-agnostic peer DIDs are used as identifiers of nodes in order to reduce the complexity and cost of resolution through external networks, as the primary objective of our study is not impacted by the DID method being used.

The payload of the invitation includes a *services* property in accordance with out-of-band protocol v1.0 to ease the burden of an explicit attachment of a DIDDoc. As DIDComm v2.0 omits this property, its corresponding receiver implementations should either refer to an attached DIDDoc in case of a peer DID or resolve via the specific DID method for the rest of the cases in order to fetch information about the sender's endpoints. Service's *recipientKeys* attribute should include at least one public key encoded in base64 since it will be used by the receiver to encrypt any further reply message. Similarly, *serviceEndpoint* serves as a mandatory field to be used as an endpoint for the encrypted response and transport of this endpoint can vary depending on the specific implementation.

This constructed invitation is finally encoded into the form of a URL before sending out via an out-of-band mechanism, thus minimizing the data in transmission as well as obscuring information in plain sight. However, its main purpose is to be served as a link that can be loaded in a browser and provides instructions on using a DIDComm agent with the invitation. If the receiver already possesses an agent, this invitation can directly be processed by the application to proceed with establishing a connection with the sender.

3.2.2 Exchange messages

With the reception of an invitation, a requester can use its information to construct a request for a DIDComm connection. It initially parses the invitation which is in URL form to fetch responder's public key and exchange endpoint. This information along with the key pair generated by the requester specifically for this relationship is used to pack a connection-request message and transmit it to the responder's exchange endpoint. Ideally, exchange and data messages should be channeled via distinctive endpoints for security reasons but however, a single endpoint is used in our prototype solely for convenience. This segregation allows the responder to carry out further filtering mechanisms for selective disclosure of the messaging endpoint and associated cryptography requisites.

```
{
  "@id":"<id of request>",
  "@type":"https://didcomm.org/didexchange/1.0/request",
  "~thread":{
    "thid":"<id of request>",
    "pthid":"<id of invitation>"
  },
  "label":"<identifier of requester>",
  "goal":"connection establishment",
  "did":"did:peer:1z8Ep5uP4rHFbVHPpNYhUYBxGPFAu4FW14rJNQ9qxZ7jkM",
  "did_doc~attach":{
    "@id":"<id of attachment>",
    "mime-type":"<media type of attachment>",
    "data":{
      "base64":"eyJwcm90ZWN.....lcvUGJxcGZ5MDZBPT0ifQ=="
    }
  }
}
```

Figure 13: Structure of a connection-request

The connection-request will be received and parsed by the responder using the key-pair generated for the invitation. While a single key-pair is used for all the invitations generated in ZeroComm, it is also viable to use separate pairs for each invitation depending on the security level required by the corresponding use-case. It is guaranteed that only the sender of the invitation is capable of sending a response since the requester's key in connection-request (which will be used for the encryption of the response) can only be decrypted by the responder.

The responder undergoes a procedure similar to the requester, in which it will generate a new key-pair and a DIDDoc for the relationship. Despite the possibility of re-using the same key-pair of invitation, it is mandatory to deploy a separate pair for the connection in order to preserve the desirable security aspects of DIDComm. The response will be encrypted with respect to the packing algorithm (as explained in 3.1) and transmitted to the endpoint of the requester.

```
{
  "@id":"<id of request>",
  "@type":"https://didcomm.org/didexchange/1.0/response",
  "~thread":{
    "thid":"<id of invitation>"
  },
  "did":"did:peer:1z4Y7prVfD2C4wjMynCopjkiWwSBY78jo537W9UtDuFYgB",
  "did_doc~attach":{
    "@id":"<id of attachment>",
    "mime-type":"<media type of attachment>",
    "data":{
      "base64":"eyJwcm90Z....eURJYzgxcGxSZTBiNExnPT0ifQ=="
    }
  }
}
```

Figure 14: Structure of a connection-response

Once this response is received by the requester, it will lookup the appropriate key-pair to unpack the message and fetch the necessary fields. At this point, both agents have shared the minimal information required for a message to be packed and sent via DIDComm protocols and hence can be considered as the conclusion of a successful DID-exchange process. However in practice, the requester will send back a *complete-message* resembling an acknowledgment of the connection-response. Figure 15 depicts the sequence of events and messages passed between the two agents during this protocol. Thread ID of the initial message corresponding to a particular DID-exchange flow is carried out through all the intermediate messages and referenced by each agent until a connection is established. Further, the corresponding regex validation for a peer DID is omitted in our implementation but should be considered in more realistic use-cases.

In the case of a peer-to-peer agent setup with this implemented prototype, the generation of invitation by the responder and the provision of this invitation to the requester are the only steps required to be done manually. The subsequent steps of the DID-exchange protocol are automated in each role such that it ultimately leads to a DIDComm connection established successfully between the agents. However, since DID-exchange is required as a prerequisite of a pub-sub relationship for establishing connections with members in a group, the invitation will be provided to the requester as an automated step in the relevant subscription process.

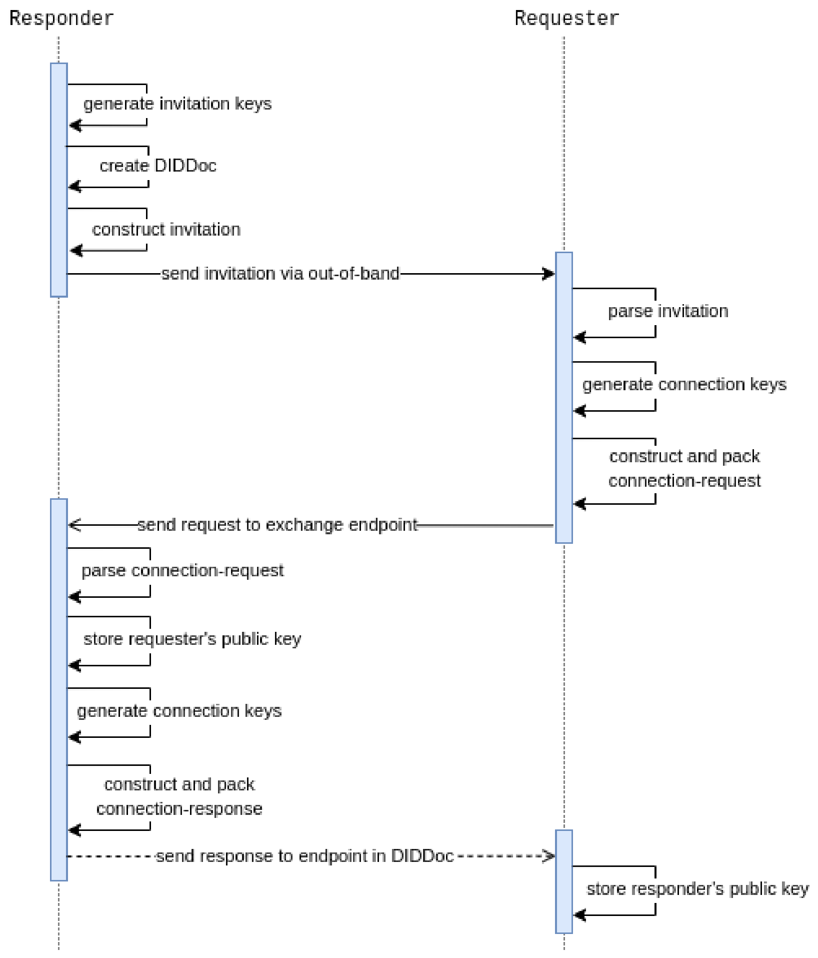

Responder
Requester
generate invitation keys
create DIDDoc
construct invitation
send invitation via out-of-band
parse invitation
generate connection keys
construct and pack connection-request
send request to exchange endpoint
parse connection-request
store requester's public key
generate connection keys
construct and pack connection-response
send response to endpoint in DIDDoc
store responder's public key

Figure 15: Sequence diagram of DID-exchange

3.3 Discover Features

As DIDComm expects to be widely adopted in diversified scenarios, it is beneficial for an agent to disclose its supported protocols such that the requesting peer can be aware of the operations in advance. As an example, a group of members can be created with a set of individual agents only if each declares compatibility with an exact group-messaging protocol. It should be noted that this discovery protocol was only considered as per the recommendation by DIF despite that it does not produce any impact on our core algorithm [22, 23].

Accordingly, a separate endpoint is included in our implementation to provide supplementary information regarding DIDComm protocols and roles of the agent, while adhering to RFC-0031 conventions [24]. In addition, ZeroComm supports querying a specific set of protocols using prefixes and wild-cards as shown in Table 1. However, responses to discovery queries should not be interpreted as negative feedback but rather a reluctance to disclose further information. For example, missing roles in an agent's response does not imply that no roles are supported for the protocol but instead, agent does not prefer to provide any specific details. Further, messages involved in this protocol are not encrypted with DIDComm since agents generally use this service prior to setting up a DIDComm connection with the corresponding agent. Therefore, best practices should be considered to maintain privacy while preventing any *fingerprinting* by malicious agents [25].

Table 1: Sample queries of discovery protocol

Query	**Response**	
	Protocol	**Roles**
*	https://didcomm.org/out-of-band/1.0	Sender, Receiver
	https://didcomm.org/didexchange/1.0	Inviter, Invitee
	https://didcomm.org/pub-sub/1.0	Publisher, Subscriber
https://didcomm.org/pub-sub/*	https://didcomm.org/pub-sub/1.0	Publisher, Subscriber

4 Design

Publisher-subscriber is a messaging pattern that transmits data among multiple parties efficiently. In contrast to the naive approach of maintaining individual connections with each other, pub-sub constructs indirect relationships based on the subjects of interest. Hence, two mutually inclusive roles are involved in the pattern known as publisher and subscriber based on the operations they perform, i.e. writing data to a subject and reading published data respectively. Message-queue data structure functions both as a storage to hold these published messages and an intermediary to connect subscribers with publishers based on their subscriptions.

This communication model can be used to resolve group messaging with multiple participants despite the lack of data security as a core feature of the pattern. In particular, Antony Rowstron et al. extended the pub-sub pattern with application-level multicasting using *Pastry* to formulate the relevant multicast tree, which supports decentralization but lacks security at the data layer [26, 27]. In contrast, we use ZeroMQ to establish the overlay network required by nodes as well as to manage data transmission with underlying message queues. Further, ZeroComm integrates data security via DIDComm while establishing trust among participants.

This chapter provides a comprehensive description of two possible solutions which entail both security and trust in the message-level of the pub-sub communication model. The pattern matching process of interested subjects can either be based on the content or a dedicated field (commonly known as *topic*) of the message [28]. However, both varieties in ZeroComm consider only topic-based subscriptions since a topic can be interchangeably referred to as a group in the context of messaging.

4.1 Requirements

We have identified a number of requirements to be addressed initially as listed below in the course of designing ZeroComm by combining pub-sub messaging pattern with DIDComm protocols.

1. DIDComm intends to be **decentralized** in terms of structural elements. This implies that agents using DIDComm should not depend on any centralized component but however,

existing pub-sub implementations usually involve external message-queue brokers apart from the individual nodes, thus conflicting with our design goal of decentralization. To this end, ZeroMQ can be used to construct a pub-sub pattern using its fundamental sockets such that each publisher or subscriber can perform without relying on any centralized module except for another peer agent.

2. DIDComm expects **end-to-end encryption** in all messages. Conversely, nodes in a pub-sub system are connected with each other based on the general relationships of interests thus being unaware of the distinctive participants. This absence of direct relationships poses a complication with regard to the encryption of individual messages.
   (a) This dilemma can be overcome by using a **shared key** for the encryption and decryption of messages specific to a particular topic. However, this approach is only secure as the weakest link since a single malicious node is sufficient to compromise an entire group of nodes.
   (b) Alternatively, each message can be encrypted with respect to **individual relationships**, thus preserving adequate security aspects in accordance with DIDComm. Nevertheless, this approach has the downside of using an excessive number of messages since each single message now needs to be encrypted for every publisher-subscriber relationship separately.
3. Relationship-wise encryption of messages as mentioned above, necessitates two additional requirements in a subscriber as opposed to the generic publisher-subscriber pattern.
   (a) **Subscriber should read only the correct message.** In contrast to the conventional pub-sub model where a subscriber is only required to match its subscription with the topic of a particular message, now it is also required to match with the corresponding encryption such that the message can be successfully decrypted and parsed.

       For example, if A and B are subscribers with subscriptions to topic t_1 with initial messages $\{m_1, m_2, m_3, ...\}$, A should only read $\{m_1^A, m_2^A, m_3^A, ...\}$ and not $\{m_1^B, m_2^B, m_3^B, ...\}$.
   (b) **For a given correct message, subscriber should know which keys to be used for decryption.**

       As an example, if A is a subscriber with public key-pairs K_B and K_C for publishers

B and *C* respectively of topic t_1, *A* should use only K_B (and not K_C) for unpacking messages $\{m_1^B, m_2^B, m_3^B, ...\}$.

This implies that each message should be identifiable with respect to its publisher which can be accomplished by different methods.

i. **Message-wise:** Identification is embodied into the messages individually (eg: HTTPS header, ZMTP frame). This approach results in more vulnerability since it fails to provide obfuscation with respect to receivers and more resource utilization by a subscriber as every message now needs to be inspected regardless of the intended member. In addition, it also imposes an unnecessary restriction on the data format of a message.

ii. **Relationship-wise:** Separate message-queues are maintained for each publisher-subscriber relationship. Since performance has already been traded off against security in DIDComm, this method is applicable in order to minimize the processing latency of a message despite the excessive usage of queues. However, this also results in an under-utilization of the overlay network constructed by pub-sub pattern as separate one-to-one communication channels are used instead of its inherent fan-out mechanism.

iii. **Fail-and-forget:** In addition to the above methods, a naive mechanism can be implemented in the subscriber where it will discard any message which fails to be unpacked with DIDComm protocols. This leads to an additional processing overhead in subscribers but nevertheless, all messages are transmitted as identical DIDComm envelopes, thus obfuscating messaging patterns along with the targeted receivers from any malicious middle-man listeners.

4. Nodes in a decentralized group must possess more or less **equal capabilities** which implies that each node should be able to function as any possible role in the messaging pattern. Hence there should not be any distinction between publishers and subscribers but instead, each should be treated as a generic member of the group in terms of the participation. This can be achieved via two different approaches.

   (a) **Lazy role based approach:** Each member is a subscriber by default and can perform publishing messages upon explicit registration as a publisher. Despite the complexity, this approach can be considered more appropriate due to the efficient use of relationships and resources.

(b) **Ambitious membership approach:** Each member is spawned as a privileged node with both read and write capabilities. This may lead to unnecessary and idle relationships among members, specifically in groups where at least one node remains only as a subscriber.

4.2 Components

An individual message in a communication model generally traverses through different phases during its end-to-end transmission process. The precise interpretation of these domain-specific layers is essential in designing the solution with respect to requirements, extending the group communication model as well as gaining insights into the overall ecosystem [29]. Figure 16 depicts the composition of ZeroComm components layered in a stack-based on such interactions and precedence.

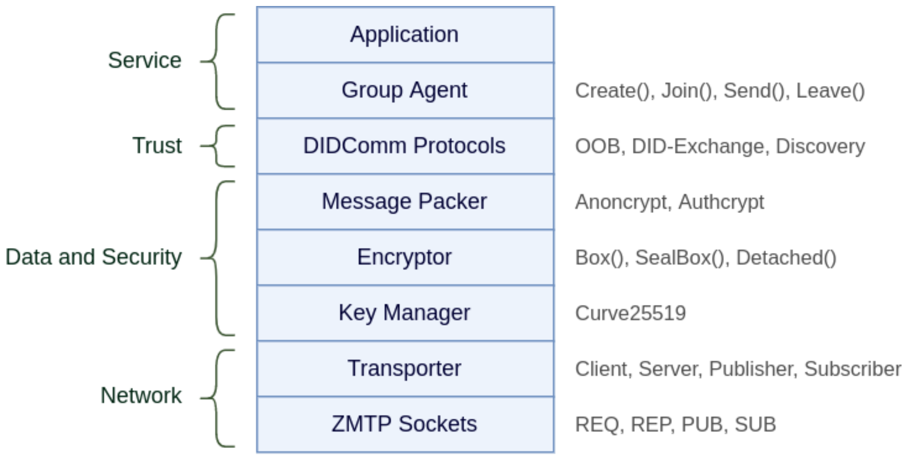


Figure 16: Stack of components

- **Application Layer**
  The topmost layer of our solution serves for any application to use the abstract functions of a group agent, particularly group communication, in order to provide higher-level services such as messaging, data streaming and event notifications. However, the current implementation does not include any specific application as we only focus on providing the underlying core communication model.

- **Group Agent**
  This layer performs as a bridge between the publisher-subscriber pattern and existing

DIDComm protocols in order to provide an interface to higher level use-cases. Specifically, key functionalities of a group member are abstracted with concrete implementations in this component such as join, leave and send a message. Both trust and security are preserved throughout these entire processes while exploiting the underlying DIDComm features.

- **DIDComm Protocols**
  A number of protocols have been introduced in the context of DIDComm but however, only the fundamentals such as *Out-Of-Band*, *DID-Exchange* and *Discovery* are considered for this model to maintain both simplicity and sufficient trust levels. In addition, our proposed solution includes v1.0 protocols despite that the v2.0 collection has already been released by DIF [23]. Hence, any future work may consider either integrating novel or upgrading existing DIDComm protocols as required.

- **Message Packer**
  ZeroComm uses this layer to pack and unpack group messages adhering to DIDComm standards as similar to the prevailing p2p implementations. However, we only focus on *authcrypt* in this solution, which is the most common packing algorithm of DIDComm. Nevertheless, *anoncrypt* algorithm can also be easily plugged into ZeroComm by satisfying our generic *Packer* interface. Each message from DID-exchange to group-communication undergoes a packing procedure thus entailing security as a facet of the message itself throughout the entire solution. This implies that higher layers in the stack are oblivious to the encryption procedure while lower layers (excluding Encryptor) are unaware of the message content and independent of the application being used.

- **Encryptor**
  This serves as an abstract layer of *NaCl* cryptography functions with respect to v1.0 DIDComm protocols for any encryption or decryption of data required by the packing algorithm. Specifically, our implemented solution supports *crypto_box_easy*, *crypto_box_seal* and *chacha_detached* functionalities exposed through this component.

- **Key Manager**
  The message packing process uses distinctive keys per each relationship for encryption of messages. Hence the necessity of a key-manager is vital with capabilities such as

generation of *Curve25519* key pairs to proceed with *NaCl* functions, storage of the generated keys and querying by the relationship whenever required to proceed with packing and unpacking functions.

- **Transporter**
  Since we intend to construct a group-communication model atop the desirable trust aspects of DIDComm protocols, ZeroComm consists of both publisher-subscriber and client-server patterns where the latter will be used only for the initial setup of DIDComm connections between members in a group. However, we also exposed explicit peer-to-peer communication to assist with any future work related to the underlying ZeroMQ transport. Transporter is agnostic of the higher layers and exploits the underlying messaging pattern for transmission of binary-encoded data.

- **ZMTP Layer**
  This layer functions as a network interface in our proposed solution where lower-level tasks such as connection establishment, transmission of individual messages, construction of overlay networks and management of message-queues are handled by ZeroMQ Message Transport Protocol [30].

4.3 Message Queues

In addition to the ordinary messages in our communication model, it also involves a separate set of messages for the propagation of state, thus coping with dynamic memberships of the group (eg: ongoing joins and leaves). Hence ZeroComm maintains a separate queue per each group to communicate these state changes of members, apart from the rest of internal queues used for the transmission of usual data messages.

Semantics of Uniform Resource Name (URN) is used with corresponding *Namespace Identifier* and *Namespace Specific String* to distinguish all queues uniquely based on the message type and relationship if applicable [31]. Thus, the syntax of a state-queue is defined as follows.

urn:didcomm-queue:<topic>:state

Based on the identified caveats in section 4.1, two different solutions can be derived in terms of the data-queue relationship constructed between publisher and subscriber. Following sub-sections describe the fundamental differences between these adopted approaches, which will be referred to as *modes* of our solution in subsequent sections. However, the distinction between two modes does not apply to state messages as we assumed group-view changes are relatively infrequent when compared to data messages and thus maintained a consistent algorithm for state changes across the two approaches.

(a) Single-mode

A single queue is used in this method in order to hold all the data messages of a group. Accordingly, the internals of pub-sub pattern including its overlay network are preserved such that all the individual messages encrypted for distinctive subscribers are published to the same topic. Further, the *fail-and-forget* mechanism described in section 4.1 is adopted instead of using separate recipient identifiers in message headers, to maintain obfuscation of messages to both internal and external parties in terms of network traffic and message content. This solution will be referred to as *single-mode* approach throughout the rest of this document. In this case, the data-queue should only be recognized with respect to the group and therefore, the syntax of a data-queue in single-mode can be formulated as follows.

urn:didcomm-queue:<topic>:data

Despite the simplicity of the architecture, this leads to more processing overheads in subscribers as they are required to read and attempt with unpacking all the messages conforming to DIDComm protocols, even if the message is intended for a different subscriber. Figure 17 depicts how queues are constructed in single-mode with proper naming syntax where P_1, P_2 are publishers and S_1, S_2 are subscribers of topic T_1.

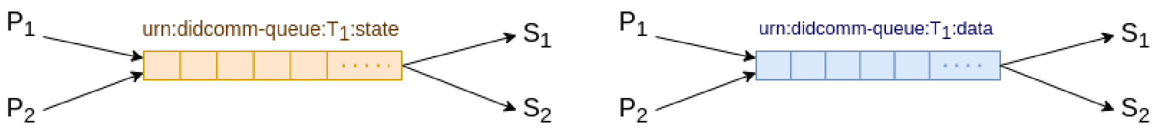


Figure 17: Single-mode queues

(b) Multi-mode

In this alternative approach which is known as *multi-mode*, the generic structure of pub-sub pattern is slightly altered to maintain separate queues per each publisher-subscriber relationship.

Furthermore, these associations are retained as simplex due to the reasons such as simplicity and efficient usage of resources in the absence of full-members (eg: a group with at least one read-only member). In essence, this requires the syntax of a data-queue to include both the direction and participants of a specific relationship in addition to the type of message.

urn:didcomm-queue:<topic>:data:<publisher>:<subscriber>

Figure 18 illustrates an example of constructing message-queues with proper naming conventions where P_1, P_2 are publishers and S_1, S_2 are subscribers of topic T_1. In terms of data messages, *multi-mode* uses a fan-out mechanism in which each individual message is packed and sent *n* times where *n* is the number of members in the group excluding the publisher. This decouples relationships even within the same group and preserves data privacy from the encryption process to the dissemination of messages. In contrast, a single message queue is used for the propagation of state updates as it is infeasible and unnecessarily complicated to maintain relationship-wise data structures for infrequent state changes.

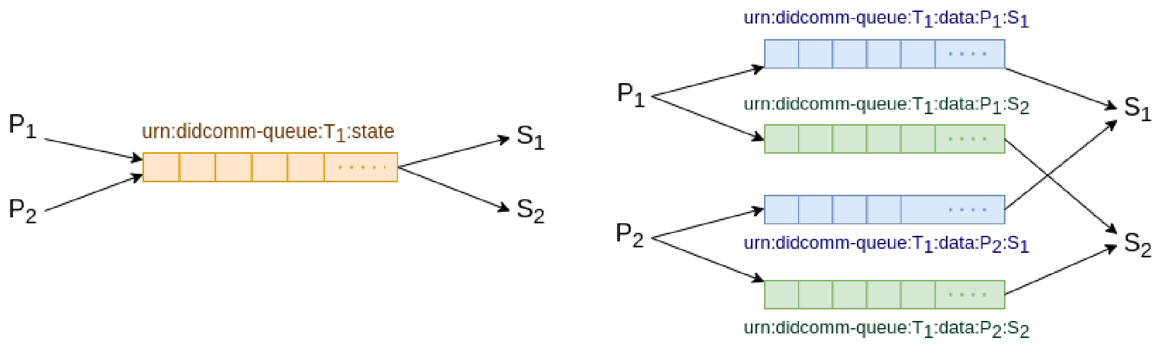

P1
P2
urn:didcomm-queue:T1:state
S1
S2
urn:didcomm-queue:T1:data:P1:S1
urn:didcomm-queue:T1:data:P1:S2
urn:didcomm-queue:T1:data:P2:S1
urn:didcomm-queue:T1:data:P2:S2

Figure 18: Multi-mode queues

4.4 State Dissemination

As ZeroComm relates to a group of members, state management is deemed crucial to support its ongoing operations efficiently, securely and without any disruption. The decentralized nature of our solution necessitates each state of a member node to be disseminated among all the rest of the members in contrast to a regular group model, where it is managed by a dedicated external component. Similar to data messages, states should also adhere to a fan-out mechanism by which each single status update is encrypted separately for every member thus imposing granular-level security even within the same group. The propagation of status can be achieved via *gossip protocols* but however, *multicast* was used instead since the solution's core model is already devised with pub-sub pattern and delayed state-changes by gossiping can often lead to more complexity, unreliability, resource over-utilization and security vulnerabilities. Thus, we considered several approaches as described below in order to achieve this state dissemination via multicasting.

1. **Separate messages with a single status topic**
   This approach encrypts status updates separately for each member and publishes the resulting *m**(*n-1*) messages to a single status topic where *m* is the number of status updates and *n* is the group size including the publisher. Since all messages are published to the same topic, an identification mechanism to correlate each message to its intended recipient is hence required. To this end, the receiver's DID can be used in a dedicated ZeroMQ frame such that a recipient does not need to decrypt the content-frame if the corresponding DID is not matched. However, as peer DIDs are used as identifiers in our solution, this can lead to an exposure of service information in addition to the excessive number of bytes caused by including peer DIDs in each message. Alternatively, a separate UUID which is known only to the parties involved such as exchange-ID in DID-Exchange protocol, can be used for message correlation thus preserving both security and efficiency. Nevertheless, this still leads to an excessive number of messages in a single state-queue.

2. **Separate state-queues per each relationship**
   In contrast to publishing all distinctively-encrypted messages to the same status topic, this approach transmits messages to separate queues. During the design phase, we based our model on an assumption which implies that realistic systems will incur less-frequent state-changes relative to data messages. Hence this approach may construct numerous message-queues unnecessarily solely for the transmission of infrequent status updates. However, this will also reduce the processing overhead of a member due to retrieving only relevant messages and thus eliminating the validation process and any redundant unpacking invocation.

3. **Encrypt using a shared-key and publish to a single status topic**
   This can be considered as the simplest as well as the most-efficient approach in terms of both resource utilization and latency. Specifically, status updates can be encrypted

via a symmetric-key as opposed to the DIDComm packing algorithm thus using only one message per update. Regardless of the reduction in processing overhead of both subscriber and publisher, this can potentially lead to insecure circumstances by exposing group information as the entire metadata will now only be secure as the weakest member. Any rectification on such cases will relatively be burdensome as well, due to the use of a shared-key and complication in isolating the malicious member.

4. **Single message with sub-sections for each member**
   In this approach, a state-change will still be packed distinctively per each intended member but however, all the DIDComm messages corresponding to a single update will be transmitted using only one message. In particular, a state message is now composed of a *map* object indexed by an identifier of the recipient (eg: exchange-ID) followed by the encrypted content of the status-change. This results in less number of messages but the size of a single message will be relatively high. However, network traffic can be expected to be lower than sending individual DIDComm messages even if they sum up to the same size as in this approach.

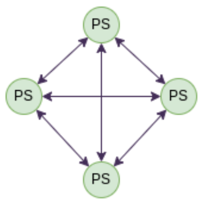
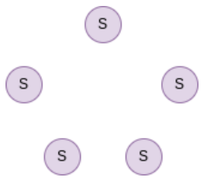
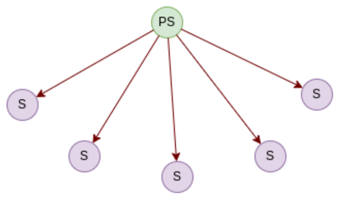
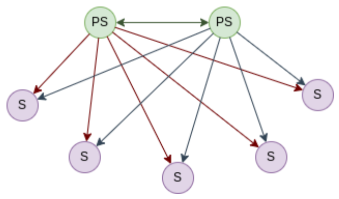
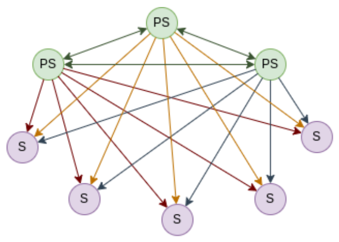
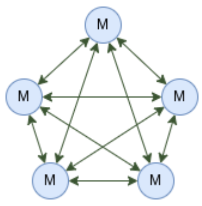
5. **Publish state messages to data topics**
   Additionally, a naive approach of publishing individual DIDComm state messages to the corresponding recipients' data topics can also be considered. As opposed to all the other alternatives, this entirely eliminates the necessity of a separate state-queue while preventing the redundant unpacking invocations at the recipient as well. This however leads all data relationships to be duplex by default regardless of the role involved, due to the bi-directional nature of state updates between two participants. Furthermore, it requires data and state messages to be coupled and thus may impose unnecessary complications in tracing and debugging separate event logs.

Regardless of the numerous alternatives, we have considered using the *sub-section* method in both *single* and *multi* mode approaches of ZeroComm. Particularly, the content of a status message is packed separately for individual members and then composed together in a single *map* object. These *authcrypt* sub-messages are indexed by the *exchange ID* used to establish distinctive DIDComm connections, thus preserving the privacy of the intended member.

4.5 Overlay Network

The network structure of a decentralized group varies along with the connectivity among members which is established based on the message type and composition of roles involved. Table 2 categorizes different possible types of graphs along with the example groups and constrained scenarios where they can occur. Read-write capability is omitted in state-queue cases as each member will operate both as a publisher and a subscriber. In addition, this analysis assumes that a group is consisted of at least 2 members.

Table 2: Categories of group networks (*S*=Number of read-only members, *PS*=Number of read-write members, *M*=Number of group members)

Message type	Constraint	Type of graph	Example
Data	*S*=0 and *PS*>1	Bidirected complete	
	S>0 and *PS*=0	Null graph	
	S>0 and *PS*=1	Unidirected acyclic disconnected	
	S>0 and *PS*=2	Directed acyclic disconnected	
	S>0 and *PS*>2	Directed cyclic disconnected	
State	*M*>0	Bidirected complete	

5 Implementation

5.1 Architecture

The proposed solution is implemented in Golang[11] which is a high-level open-source programming language with inherent accommodation for concurrent and high-performant software systems. In addition, Python[12] and Shell[13] scripts are used together for automation, testing, data-processing and visualization purposes. ZeroMQ v4.0 is used as the transport provider and to fulfill its underlying network requirements via TCP socket connections. The final version of the implementation can be found at the Github repository[14] with 9000+ lines of code.

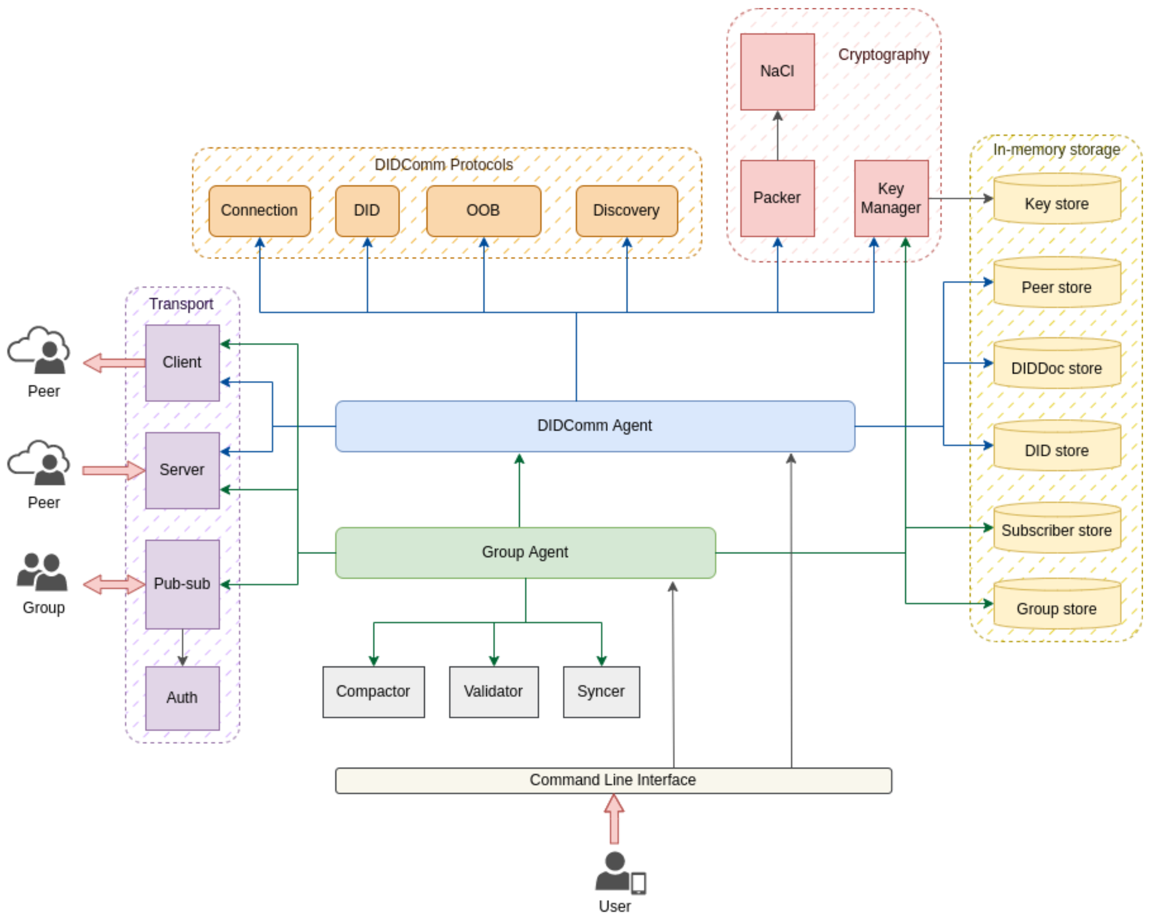


Figure 19: Software architecture

[11]Also referred to as Go (more information is available at https://go.dev/doc)
[12]An interpreted, object-oriented and high-level programming language
[13]A scripting language for Unix-based operating systems
[14]Source code is available at https://github.com/YasiruR/didcomm-prober

The core functionalities of the solution with respect to group member, DIDComm agent, message packer, transport service, key-manager and other vital services are abstracted through interfaces to serve both as a convenient guidance of the implementation and a decoupling mechanism for any modification of the code. Figure 19 provides an abstract view of the internal architecture and how components interact with each other. A short description of each key element is given below along with dependent sub-components.

- **Group agent**
  This can be regarded as the most significant component of ZeroComm as it includes all the concrete flows of a group member to support the basic operations, which will be described in section 5.2. As shown in Figure 16, group agent utilizes a number of DIDComm protocols that are provided by the DIDComm layer such as invitation-related functions and individual connection establishments. In addition, three localized components are used in a group agent known as *compactor*, *validator* and *syncer* in order to compress group-state messages, carry out checksum validation upon joining a group and maintain Lamport timestamps in data-messages, respectively. In essence, these are supplementary services of ZeroComm that enhance the core algorithm's desirable properties such as performance, consistency and ordering of messages.
  The transport requirements of a group member are satisfied by ZeroMQ, through which multiple listeners are initiated to process the different types of messages involved in our solution. Further, an authenticator is used at the transport-level in order to provide authenticity and restrict any unnecessary socket connections which have not been established through an initial DIDComm handshake. In order to both preserve simplicity in our Proof-of-Concept as well as to enhance security aspects, we used in-memory storage instead of persistence. Hence two thread-safe stores are initialized as *subscriber* and *group*, to assist with unpacking messages of subscribed topics and retrieval of group information whenever required.

- **DIDComm agent**
  This component serves as an individual agent which uses DIDComm to establish secure connections and transmit messages with connected peer agents. As described in section 4.2, this resides in a lower level than the group agent and should not be impacted by the higher layers. In fact, a group member is an individual DIDComm agent with extended

functionalities and multiple peer-to-peer relationships to support group communication. Hence this layer interacts with other internal components such as DIDComm protocols, message-packer including a wrapped encryptor around *NaCl* cryptography library and multiple stores for peer information, DIDs and DIDDocs. However, it only exploits the client and server sockets of ZeroMQ for all the underlying functionalities, since this layer is not aware of any higher-level applications such as group messaging.

- **Key Manager**
  Both generation and storage of cryptography key-pairs in ZeroComm are carried out by a separate component along with the capability of queries. This is a vital dependency to provide secure communication in our solution as each message will be encrypted differently based on the recipient. We only considered a naive implementation with in-memory stores but however appropriate security measures with best practices should be followed to enforce further restrictions on the end device, which will be discussed in section 7.1.4.

5.2 Functionality

This section describes higher-level abstractions of a group agent which were implemented in ZeroComm to support secure communication among multiple parties and hence each message of this layer is transmitted only through DIDComm. Both *single* and *multi* modes adhere to the same set of algorithms apart from the variations in topic names and subscriptions as described in section 4.3. We will further discuss the design choices of ZeroComm in section 7 with a comprehensive reflection on the corresponding trade-offs.

Since our solution is a Proof-of-Concept, we assumed that the network is reliable and hence acknowledgments can be ignored in order to spotlight the core algorithm as well as to keep the implementation less complicated. Therefore, the functions described below may omit the final round-trip of messages which should be considered in more realistic scenarios with unreliable networks.

5.2.1 Create

This function constructs a new group uniquely identified by a group-name which is analogous to a topic in the pub-sub messaging pattern and members are not allowed to create multiple groups with an identical name. In order to preserve decentralized sovereignty, ZeroComm does not permit privileged users including the creator of a group. During the *create* process, the respective node will store itself as a member along with attributes such as label, status (active/inactive), role in the group (read-only/read-write), an invitation for DID-exchange, the publishing endpoint of messages and a generated checksum for the group. All this information is mandatory for operating as a member of the group and hence fetched in both create and join functions. The generated checksum value will be updated regularly per every subsequent group-view change and also maintained by each successive member separately.

5.2.2 Join

The joining process can be regarded as the crux of ZeroComm due to the majority of threats and vulnerabilities of a group can be exploited if this functionality is not properly designed. The flow will be initiated by a joiner (*requester*) requesting from an existing member of the group (*acceptor*) via a DIDComm message. Hence, a DIDComm connection should prevail in advance between requester and acceptor as the only prerequisite of the flow. Subsequent DIDComm connections with the rest of the group will be established in the course of the procedure, if not already established.

Figure 20 depicts the flow of events that occur during a join process. Acceptor's DIDDoc which is shared with the joiner should contain a group-join service endpoint through which the joiner will be validated and receive further contact information of the residual members to establish individual DIDComm connections (prerequisites of DID-Exchange protocol as described in section 3.2). This retrieved group-view will be stored and updated in joiner to become an acceptor and accommodate any future requests by new members to join the group. Each member possesses the ability to act either with *read-only* or *read-write* privileges upon registration as a member (both when creating and joining a group).

A subscribe request is transmitted to each member separately via DIDComm once a connection is established with the member. This particular subscription serves as a higher-level abstraction of all the underlying steps to create a secure pub-sub relationship including connecting and binding network sockets, subscribing via ZeroMQ topics, exchange of public keys, setting up authentication at the transport layer and storing peer information for the retrieval of future messages. When this subscription process terminates successfully, the requester will be con-

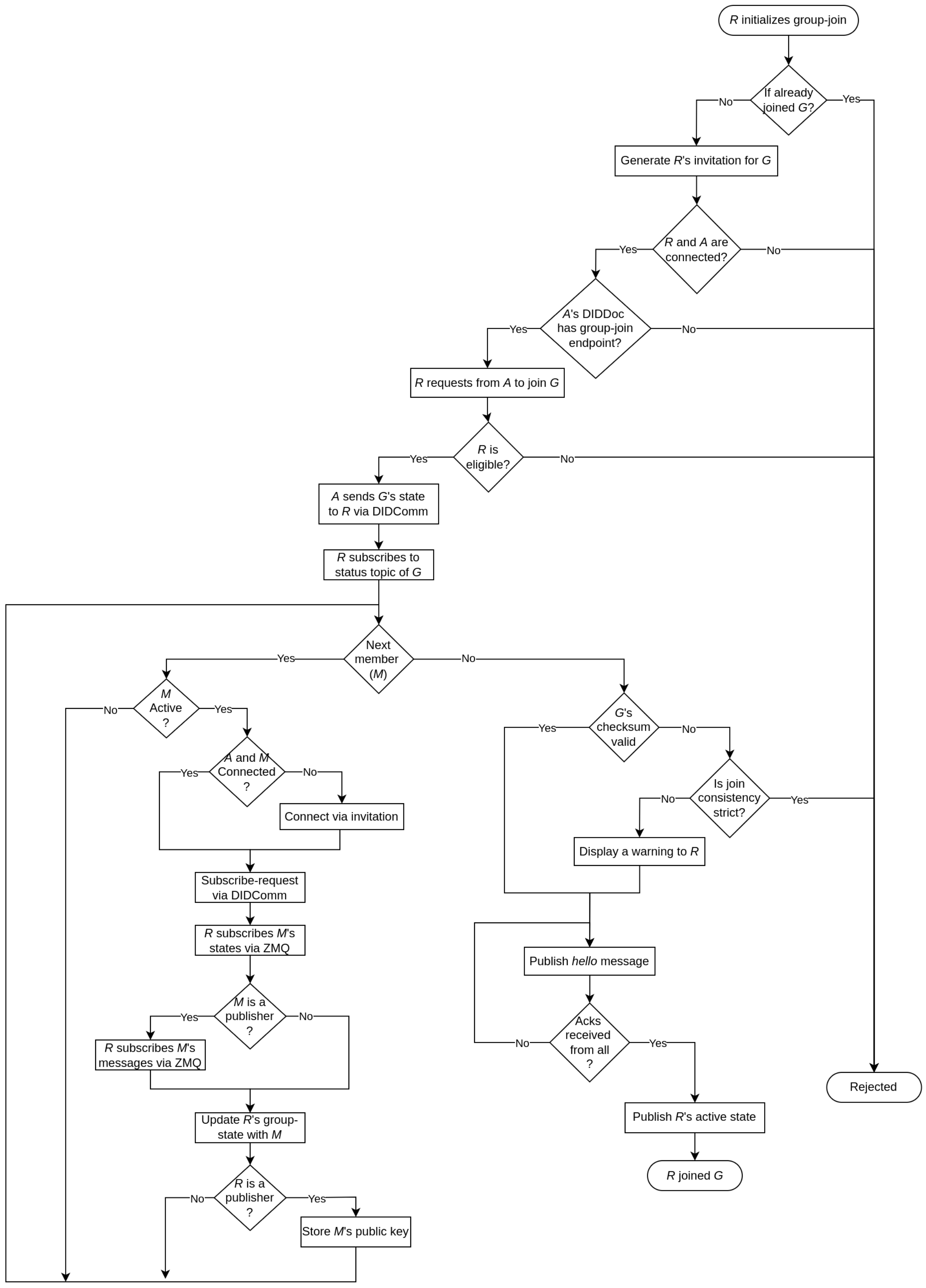


Figure 20: Flow chart of join process (*G*=Group, *R*=Requester, *A*=Acceptor, *M*=Member)

cluded as a member in the adjacent peer's local view of the group along with the relationship required between the two peers for group communication.

In the context of ZeroMQ, transmission of data messages is handled in the background with an asynchronous I/O model whereas a subscription takes non-zero latency for setting up the connection through a TCP handshake. This overlap between connection and transmission leads to a possibility of missing an initial set of messages sent by the publisher until the subscriber is connected successfully. The aforementioned dilemma can be resolved naively by delaying the publisher with a defined latency buffer such that all socket connections are established prior to the transmission of any message by the publisher. However, our solution consists of a more adequate approach in which it publishes a stream of encrypted *hello* messages with a configured time interval and proceeds with publishing ordinary messages only after the corresponding acknowledgements from all group members are received.

A joiner undergoes a series of validations at three different stages: during the initial DID-Exchange with acceptor, when processing the group-join request and during DID-Exchanges with other members. This final set of individual validations allows any particular member to prevent from connecting with a distrustful joiner even if the joiner is connected with the rest of the group. Hence, it enables a granular level of connections within a group, possibly leading to inconsistent views among members. In particular, a joiner will gather group-checksum values from each member during the join process and validates them when there are at least two members excluding the joiner. This poses a complication in distinguishing byzantine nodes from trustful members with only conflicting views but nevertheless, it was designed to allow any subsequent joiner to also proceed with a warning as our work focuses on preserving decentralization with possible intruder detection and not on recovering from byzantine members. However, a group can also be configured to restrict joins when the virtual synchrony across the group seems to be inconsistent.

5.2.3 Leave

As ZeroComm is decentralized and group states are maintained in all individual nodes, any update to the group should be notified to all the members. This implies that leaving of members from the group should be carried out in a graceful manner such that each local state in members is updated accurately. To this end, a notification is published to the status topic and subsequently, the corresponding ZeroMQ subscriptions are discontinued. Finally, the local storage of leaver will be discarded apart from the individual DIDComm connections as they operate at a lower peer-to-peer level in our component stack and should not be impacted by a group action.

5.3 Messages

ZeroComm transmits different message types among members in order to fulfill the basic requirements of a group communication model. Accordingly, each agent initiates a number of listeners by means of ZeroMQ sockets to retrieve and process these messages independently from each other. This section provides a description of such listeners with respect to the involved message type.

5.3.1 Join requests

Since existing members should support adding newcomers to the group, a listener is initialized in an agent to process any future join request. The information about this listener is conveyed through the agent's DIDDoc with a separate service endpoint named *group-join*. Once a join-request is received by the endpoint, sending agent's role will be defined as the *joiner* or *requester* while the receiver will be known as *acceptor*. This handler uses a *REP* socket and should include a validation to verify if the joiner is eligible to be aware of the group information. Our current implementation contains only a dummy validation but however, more adequate and complicated procedures should be followed as necessary.

5.3.2 Subscriptions

Any subscribe request transmitted between the members of a group is handled by a separate listener. As similar to the join-request handler, this also utilizes a *REP* socket and includes a validation to provide the granular-level choice of connection establishment with the sender of a subscribe message. In addition, this paves the way to initiate a pub-sub relationship using the corresponding ZeroMQ sockets as well as enables transport-level authentication between the two parties involved.

5.3.3 Status

In order to facilitate the necessity of transient memberships, a separate listener is initialized for updating local view of the group individually in members per each state-change. These updates may include alterations to invitation, label, publish-endpoint, active status or role of a member. A *SUB* socket is used for the retrieval of messages and this resultant fan-out mechanism lessens the complexity of disseminating member-updates to the entire group.

```
{
    "@id": "f016777b-07ad-49f2-bf9b-c2e02066c991",
    "@type": "https://didcomm.org/pub-sub/1.0/status",
    "topic": "test_topic",
    "auth_msgs": {
        "c3d11afc-84fc-4a37-8d8f-f7e5b118629e": {
            "protected": "eyJlbmMi.......T0ifX1dfQ==",
            "iv": "NDMyNDc0NTQ4MzgzODE4Mjg3Mw==",
            "ciphertext": "5u+3DdB......edF+AdW0=",
            "tag": "oC9OgXCV8URkCl45ReO2qQ=="
        },
        "c4bd9ade-61bb-4cb6-aff0-73d6b89e6f6c": {
            "protected": "eyJlbmM......ifX1dfQ==",
            "iv": "MjcwMzM4NzQ3NDkxMDU4NDA5MQ==",
            "ciphertext": "YP5Wl+xw......jDWhO/TQ=",
            "tag": "udMzCol8uqvSdP0gaF5dLA=="
        }
    }
}
```

Figure 21: Status message with 2 members

A single state-message consists of sub-sections for each member in the group (Figure 21) with respect to DIDComm protocols as described in section 4.4. Therefore, a validation is included in the handler to extract the exact sub-section which is relevant to the recipient. Privacy of the intended member is preserved in this case, as it is only indexed by the *exchange ID* which does not provide any insight to an outsider except for the 2 peers involved.

Regardless of the security and simplicity, this composed data structure leads to a scalability issue where message size grows with the number of recipients. A *lossless compression* algorithm can be used in order to reduce the overall message size, which can potentially be noteworthy for larger group sizes. To this end, *zstd* seems applicable as it is currently known for lower latency and higher compression ratio compared to other existing algorithms such as *zlib* and *brotli* [32]. For example, state-message of a new member joining a group-size of 25 induces a message with an average size of 41.268kB which is then compressed into an average of 26.654kB using *zstd* algorithm, thus reducing the memory required per single message by 35.41%.

5.3.4 Data

A data handler is bound to a separate *SUB* socket to accommodate the regular messages published to a group. However, the number of messages received by this listener may vary depending on the mode (*single*/*multi*) used in our solution. Upon retrieval of a message, it will be parsed and unpacked per DIDComm protocols since group messages in ZeroComm are constructed atop the fundamental DIDComm layer.

5.4 External dependencies

Table 3 consists of the external libraries and their corresponding versions which were used in the implementation of our proposed communication model while omitting the builtin *golang* packages.

Table 3: Libraries and versions

Library	Version	Description
pebbe/zmq4[15]	v1.2.9	*Go* interface to ZeroMQ version 4.0 with *CGo* bindings around the core-implementation
klauspost/compress[16]	v1.15.14	Optimized *Go* compression packages including *zstd* compression algorithm
google/uuid[17]	v1.1.1	*Go* package for UUIDs based on RFC 4122 [33] and DCE[18] 1.1: Authentication and Security Services
btcsuite/btcutil[19]	v1.0.3	Provides bitcoin-specific convenience functions inclduing a *base-58* encoder for DIDComm message-packaging
GoKillers/libsodium-go[20]	v0.0.0	A complete overhaul of the *Go* wrapper for *NaCl* cryptography functions

[15]ZeroMQ library (available at https://github.com/pebbe/zmq4)
[16]Compression library (available at https://github.com/klauspost/compress)
[17]UUID generator (available at https://github.com/google/uuid)
[18]Distributed Computing Environment
[19]Encoder (available at https://github.com/btcsuite/btcd)
[20]Cryptography library (available at https://github.com/GoKillers/libsodium-go)

6 Evaluation

This chapter provides a comprehensive description of the experiments carried out to evaluate the performance of ZeroComm. In particular, first sub-section discusses the metrics and experiments required for assessing key-design choices of our implementation while the latter sections include the measured results of experiments, followed by an extensive rationale for the observed patterns and deviations.

6.1 Metrics and Operations

6.1.1 Joining

In the course of our implementation, we identified that the joining process of a group member plays the most significant part of the solution in terms of vulnerabilities, processing overhead and complexity. During the design phase of the join algorithm, we traded off performance to provide a highly-secure system by using exhausting packing algorithms for every message preceded by a number of DIDComm handshakes each with multiple round-trips of messages. Hence our primary objective is to evaluate how well the model performs against an impactive set of independent variables with the designed join algorithm. To this end, we identified a number of possible experimental variables as given below.

- **Initial group size:** Due to the fact that the joining process involves several DIDComm messages transmitted back and forth by a joiner with all the individual members, initial size of the group at the time of join can be regarded as a highly significant factor.

- **Initial connectivity with members:** As part of the joining procedure requires members to be connected via DIDComm, the existence of individual DIDComm connections prior to a group-join may influence the overall latency of the algorithm. For a particular group, this will result in *n* number of additional test cases with respect to the number of already connected members, where *n* is the initial group size excluding the joiner. Nevertheless, our evaluation will focus only on the two extreme cases to recognize the impact of the parameter, i.e. when the joiner is initially connected with every member vs when it is only connected with the acceptor.

- **Join consistency:** Our model provides a configurable attribute to support virtual synchrony with consistent group views via a checksum validation during the joining process. However, as the implementation only provides a mechanism for inconsistency detection and does not include any subsequent action on recovery, the configurable variable of consistent joins will not contribute any impact on the algorithm's performance.

Since our experiments should focus on measuring the performance, time consumed for the successful completion of a single joiner against varying constraints as listed above can be considered as a key-metric of the assessment. Moreover, it can be extended with a set of throughput experiments where concurrent join-requests are initialized by distinctive members simultaneously, in order to simulate a more realistic scenario and examine the model's execution in such cases. However, for the latter experiment, requesters were allowed to join groups with inconsistent views due to the inherent nature of parallelism in the experiments. Thus throughput and latency can be regarded as two key metrics of our solution as analogous to the evaluation of prior work related to pub-sub systems [34].

6.1.2 Messaging

Despite our major contribution being fixated on the join algorithm, sending group messages among members can still be considered as the key feature of a group communication model. Further, our proposed solution partakes in a computationally-intensive packing procedure per each message, after which they are transmitted differently through queues depending on the configured mode of the solution (*single* or *multi*). Hence, this necessitates a set of experiments to investigate the processing overheads incurred by cryptography functions as well as the model's performance in terms of publishing group messages with respect to the two implemented modes.

Similar to the join operation, we defined latency and throughput as the key metrics of group-message experiments. However, in addition to the group size to which a message will be published, the configured mode of the group also serves as an independent variable in this set of experiments. Since ZeroMQ does not guarantee a reliable message delivery by default, we also considered the success rate of published messages as an affirmation of our results as well as to gain further insights on any fluctuations. As discussed in section 7.3, ZeroComm only provides the data infrastructure required to enforce ordering among messages by maintaining and attaching Lamport timestamps in each message. Specifically, we assumed that our core messaging model should not be influenced by concrete ordering mechanisms but rather be defined within the application layer by means of the attached timestamps. Hence, the ordering of messages can be neglected in this evaluation as it does not impact the performance of ZeroComm.

In addition to the performance-related metrics, it is worthwhile to evaluate the resource utilization as we traded off efficiency in ZeroComm to achieve our security goals. To this end, transformation in number of bytes due to the imposition of DIDComm packing protocols can be regarded as applicable since it reflects the additional resources as well as network overhead bound with each message. In contrast to a data message, ZeroComm executes a compression algorithm on a status to overcome the inherent issue of oversized messages with sub-sections. Therefore, it can be beneficial to experiment with state messages separately such that it provides insights into the effectiveness of the compression algorithm.

6.2 Experiments

6.2.1 Environmental setup

The experiments were set up in Microsoft Azure Cloud Platform with general-purpose server nodes configured as shown below in Table 4. Each VM (Virtual Machine) was installed with Linux Ubuntu 20.04 Operating System and equipped with a standard SSD (Solid State Drive) hard-disk.

The server instances were located in 6 different time zones (Figure 22) in order to simulate the experiments as identical to a real-world scenario resembling the dispersed individual DIDComm members of a group. Each experiment was triggered by one or few test agents hosted in the same virtual machine while 5 distinctive server nodes were spawned in every other time zone. However, multiple peer instances were bootstrapped uniformly among the existing virtual machines for test cases with initial group sizes larger than 25. For example, alice, alice-1 and alice-2 belonged to the same geographical location while alice-1, alice-12 and alice-13 were initialized in the same virtual machine.

Peerings were enabled between each pair of virtual networks such that individual nodes were capable of communicating via internal IP addresses. Such peered channels route directly

Table 4: Specification of server nodes

Agents	Location	CPUs	RAM	Archi.	Max. IOPS
tester-1, tester-2, ...	Washington, USA	4	16GiB	x64	2880
alice, alice-1, ..., alice-4, alice-02, ..., alice-42, alice-03, ..., alice-23	Amsterdam, Netherlands	2	4GiB	x64	1280
bob, bob-1, ..., bob-4, bob-02, ..., bob-42, bob-03, ..., bob-23	Hong-Kong				
carol, carol-1, ..., carol-4, carol-02, ..., carol-42, carol-03, ..., carol-23	Johannesburg, South Africa				
david, david-1, ..., david-4, david-02, ..., david-42, david-03, ..., david-23	Virginia, USA				
eve, eve-1, ..., eve-4, eve-02, ..., eve-42, eve-03, ..., eve-13	San Antonio, TX				

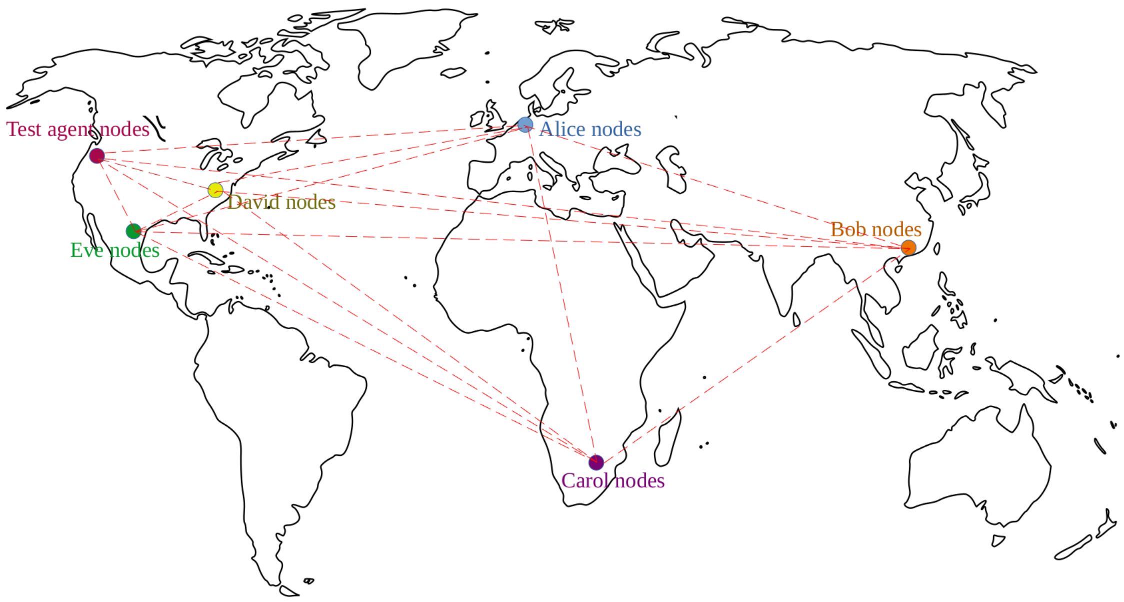


Figure 22: Geo-locations of server nodes

through the Microsoft backbone infrastructure, thus eliminating the influence of the public Internet in our experiments. Hence, the model's performance on heterogeneous networks is disregarded in the evaluation but can be considered in any related future work [35, 36].

A couple of additional modifications were also applied to the system configuration of tester agents in order to carry out experiments with larger group sizes as well as higher throughput. Specifically, the SSH connection limit was raised to 65 along with the file descriptor limit to 100,000. Moreover, *libzmq* was installed in each server node via *apt* package repository as the only prerequisite for executing the implemented Golang binary of ZeroComm.

6.2.2 Results

Following tables contain the observed measurements of the metrics and experiments defined in section 6.1. We also calculated the average values of time measurements in cases where all 3 test attempts were successful. In subsequent tables, *Initial size* refers to the number of members when the test attempt was initiated while *Yes* in the *Connected* column resembles the *testers* were connected to all group members in advance. It should also be noted that logging operations of each node might have a relatively minor but non-zero influence on the results.

A separate *pinger* script was used to measure the network latency for a basic message originated by the test agent. To be more specific, three separate HTTP requests were transmitted to each node server used in the experiments and complete round-trip latency was measured by the initiator, i.e. tester node. An average value per each geographical location was then calculated using all the individual latency measurements as listed in Table 5.

Table 5: Ping round-trip latency results

Node prefix	**Average ping latency (ms)**
alice	193.00
bob	185.67
carol	406.67
david	94.33
eve	58.00

Table 6: Join latency results

Initial size	**Connected**	**Latency (ms)**			
		Attempt 1	**Attempt 2**	**Attempt 3**	**Average**
1	Yes	1243.0	1238.0	1224.0	1235.0
2	Yes	1229.0	1228.0	1243.0	1233.3
2	No	2211.0	2208.0	2206.0	2208.3
4	Yes	2516.0	2523.0	2513.0	2517.3
4	No	4710.0	4725.0	4705.0	4713.3
8	Yes	2538.0	2551.0	2548.0	2545.7
8	No	4746.0	4746.0	4752.0	4748.0
16	Yes	2703.0	2721.0	2721.0	2715.0
16	No	4945.0	4913.0	4911.0	4923.0
32	Yes	2676.0	2656.0	2699.0	2677.0
32	No	4853.0	4869.0	4930.0	4884.0
64	Yes	3192.0	3194.0	3170.0	3185.3
64	No	5388.0	5375.0	5396.0	5386.3

Table 7: Join throughout results

Initial size	**Connected**	**Test batch size**	**Latency (ms)**
4	Yes	4	2608.0
4	Yes	16	2725.0
4	No	4	8381.0
4	No	16	23022.0
16	Yes	4	2966.0
16	Yes	16	3134.0
16	No	4	8573.0
16	No	16	23596.0
64	Yes	4	3711.0
64	Yes	16	5412.0
64	No	4	9293.0

Table 8: Group-message experiment results

Mode	Initial size	Test batch size	Success (%)	Latency (ms)			
				Attempt 1	Attempt 2	Attempt 3	Average
single	1	1	100.0	76.0	76.0	75.0	75.7
single	1	10	100.0	77.0	77.0	77.0	77.0
single	1	50	100.0	362.0	354.0	353.0	356.3
single	1	100	100.0	359.0	360.0	360.0	359.7
multi	1	1	100.0	72.0	72.0	72.0	72.0
multi	1	10	100.0	74.0	73.0	72.0	73.0
multi	1	50	100.0	358.0	356.0	353.0	355.7
multi	1	100	100.0	368.0	362.0	395.0	375.0
single	2	1	100.0	78.0	77.0	79.0	78.0
single	2	10	100.0	233.0	231.0	233.0	232.3
single	2	50	100.0	376.0	374.0	377.0	375.7
single	2	100	100.0	523.0	526.0	525.0	524.7
multi	2	1	100.0	74.0	76.0	75.0	75.0
multi	2	10	100.0	77.0	76.0	77.0	76.7
multi	2	50	100.0	370.0	367.0	370.0	369.0
multi	2	100	100.0	376.0	379.0	379.0	378.0
single	4	1	100.0	157.0	156.0	156.0	156.3
single	4	10	100.0	467.0	468.0	467.0	467.3
single	4	50	100.0	1088.0	1091.0	1093.0	1090.7
single	4	100	100.0	1406.0	1409.0	1406.0	1407.0
multi	4	1	100.0	156.0	156.0	156.0	156.0
multi	4	10	100.0	159.0	159.0	159.0	159.0
multi	4	50	100.0	778.0	778.0	775.0	777.0
multi	4	100	100.0	792.0	792.0	789.0	791.0
single	8	1	100.0	167.0	160.0	160.0	162.3
single	8	10	100.0	794.0	792.0	793.0	793.0
single	8	50	100.0	1435.0	1433.0	1439.0	1435.7
single	8	100	100.0	1760.0	1758.0	1754.0	1757.3
multi	8	1	100.0	160.0	159.0	159.0	159.3
multi	8	10	100.0	464.0	463.0	463.0	463.3

multi	8	50	100.0	794.0	795.0	797.0	795.3
multi	8	100	100.0	1100.0	1099.0	1095.0	1098.0
single	16	1	100.0	786.0	465.0	473.0	574.7
single	16	10	100.0	1108.0	1105.0	1106.0	1106.3
single	16	50	100.0	1738.0	1741.0	1741.0	1740.0
single	16	100	91.47	1842.0	1841.0	1350.0	-
multi	16	1	100.0	477.0	160.0	161.0	266.0
multi	16	10	100.0	480.0	473.0	471.0	474.0
multi	16	50	100.0	1099.0	1099.0	1097.0	1098.3
multi	16	100	100.0	1419.0	1409.0	1416.0	1414.7
single	32	1	100.0	789.0	470.0	473.0	577.3
single	32	10	100.0	1717.0	1419.0	1420.0	1518.7
single	32	50	77.88	1842.0	1843.0	1837.0	-
single	32	100	58.65	1517.0	1518.0	1518.0	-
multi	32	1	100.0	475.0	158.0	166.0	266.3
multi	32	10	100.0	784.0	783.0	781.0	782.7
multi	32	50	100.0	1408.0	1411.0	1414.0	1411.0
multi	32	100	100.0	1732.0	1722.0	1730.0	1728.0
single	64	1	100.0	1114.0	769.0	780.0	887.7
single	64	10	100.0	1757.0	1761.0	1757.0	1758.3
single	64	50	57.02	1838.0	1840.0	1841.0	-
single	64	100	42.38	1840.0	1841.0	1842.0	-
multi	64	1	100.0	490.0	466.0	167.0	374.3
multi	64	10	100.0	1093.0	1091.0	1093.0	1092.3
multi	64	50	100.0	1727.0	1726.0	1725.0	1726.0
multi	64	100	99.31	1828.0	1448.0	1442.0	-

6.3 Analysis

This section intends to provide an overview regarding the performance and efficiency of Zero-Comm by analysing the observed results based on both internal and external characteristics of our implemented solution.

6.3.1 Join algorithm

(a) Latency

Figure 23 illustrates the latency results of a group-join experiment with respect to the attempts of a single new node. The first test case (*group-size*=1) has only the *connected* scenario as a joiner should at least have one initial DIDComm connection with an existing member of the group. Small-scale error bars imply that all attempts in each test scenario had resulted in the same measurements thus revealing the stability of the proposed solution.

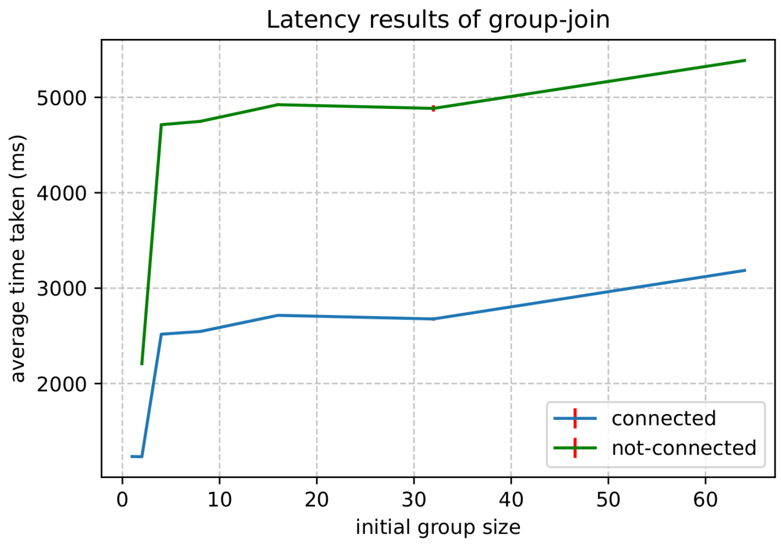


Figure 23: Latency graph of join

If the joiner is not initially connected with the entire group, the missing DIDComm connections with the rest of the members will be established in parallel as these operations do not depend on each other (Figure 24). Despite the concurrent execution, these DIDComm formations still induce a higher latency overhead due to the transmission of multiple DID-exchange messages with each unconnected member as explained previously with Figure 15. This clarifies the gap between 2 lines in the resultant graph in which *not-connected* line shows high latency values but exactly with the similar pattern as *connected*, due to the fact that the only difference between identical test cases being the transmission of DID-exchange messages. However, the different phases mentioned in Figure 24 should execute sequentially, after which the status of a joiner will be published to the group indicating the conclusion of a join procedure.

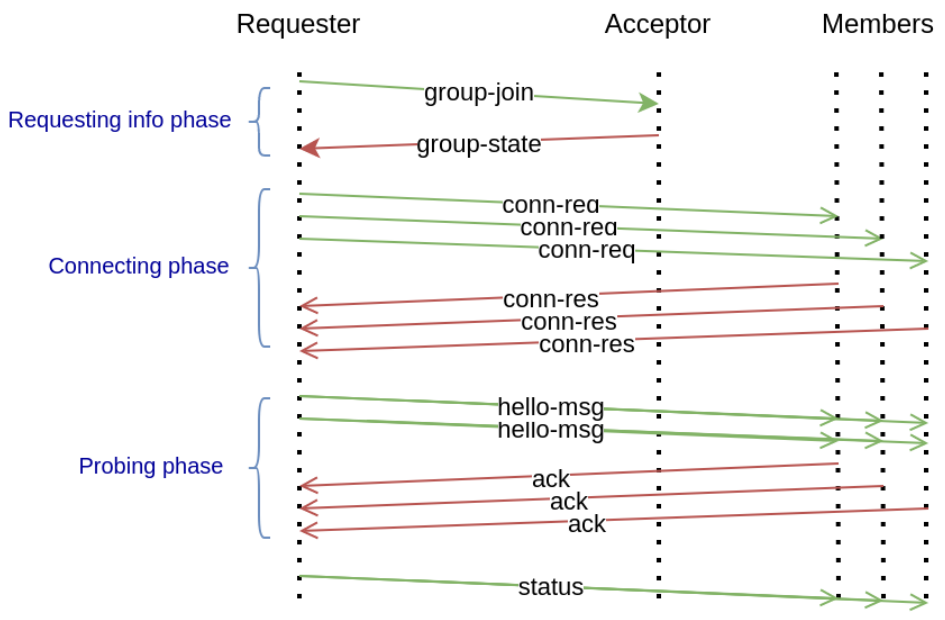


Figure 24: Messages exchanged during a join process

During the probing phase, joiner will publish a continuous stream of *hello* packets with a configured time interval (eg: 100ms), until acknowledgements from all the group members are received. This also provides an explanation to why the latency values are almost equal for the initial group size 1 and 2 in the *connected* test scenario. Specifically, *tester* in this case is probing *alice* and *bob* nodes which are located within approximately the same distance in terms of network latency (Table 5) while the rest of the messages are identical in both these cases, thus resulting in similar latency values for group-join operations.

Both *connected* and *not-connected* graph lines show a sudden increase in latency when the group size is changed from 2 to 4. This is due to the inclusion of two new members where *carol* located in South Africa shows a contrasting high network latency as shown in Table 5. Both lines then maintain approximately consistent latency values up to the group size of 32, since all members in these cases are uniformly distributed across the 5 different geographical locations. The overall latency is dominantly influenced by the upper bound latency value of each phase. This observation along with the time taken for a general join operation can theoretically be deduced from the equation given below where $t_c(u_i)$ and $t_h(u_i)$ are round-trip latency values with peer u_i for connection messages and hello acknowledgements respectively.

$$t_{join} = t_{join\text{-}req} + t_{state\text{-}res} + \max(\{t_c(u_1), ..., t_c(u_1)\}) + \max(\{t_h(u_1), ..., t_c(u_1)\}) + t_{status}$$

However, our results do not include the time taken for a state message to be received by group members as we assumed that dispatch of the subsequent status from a publisher can be regarded as the conclusion of the corresponding join process.

Both *connected* and *not-connected* graph lines show a gradual rise in latency when the initial group size is increased up to 64. In both these cases, the joiner is waiting for acknowledgements from all 64 members for a successful termination of the experiment. Due to the variation in network latency among members, this may possibly lead to the transmission of multiple *hello* messages as well as retrieval of multiple acknowledgements from the same set of members continuously until an acknowledgement is received from the last member. Subsequently, underlying ZeroMQ sockets and message-queues can be overwhelmed with network traffic and therefore, the overall latency of join operation for cases with higher cardinality of group members can be expected to be directly proportional to the group size with a slightly increasing gradient.

(b) Throughput

In this experiment, a burst of join requests defined by *batch* was sent out to the members of a group by distinctive test agents and the aggregated time taken for the successful termination of all join processes was measured. Acceptors in each test attempt were defined in advance such that the set of join requests was transmitted uniformly among the group. The experiment was carried out for initial group sizes of 4, 16 and 64 with both *connected* and *not-connected* scenarios, except for *batch*=16 since its *group-size*=64 and *not-connected* test case resulted in an over-utilization of resources along with an excessive latency.

As shown in Figure 25, the overall time taken in each case has a linear relationship with the initial group size. If we consider *batch*=4 and *connected* test scenario, only the probing phase (Figure 24) is impacted by the initial group size. More specifically, requesting information phase has hardly any influence due to the fact that group-join requests and responses involve mutually exclusive member sets (eg: *tester-1* requests from *alice-1*, *tester-2* requests from *bob-1*...) while they are transmitted and processed concurrently as well.

However, there exist several other factors within a single agent which can possibly lead to a latency increment in cases of concurrent joiners. Synchronization with *mutexes*, *atomic-counters*

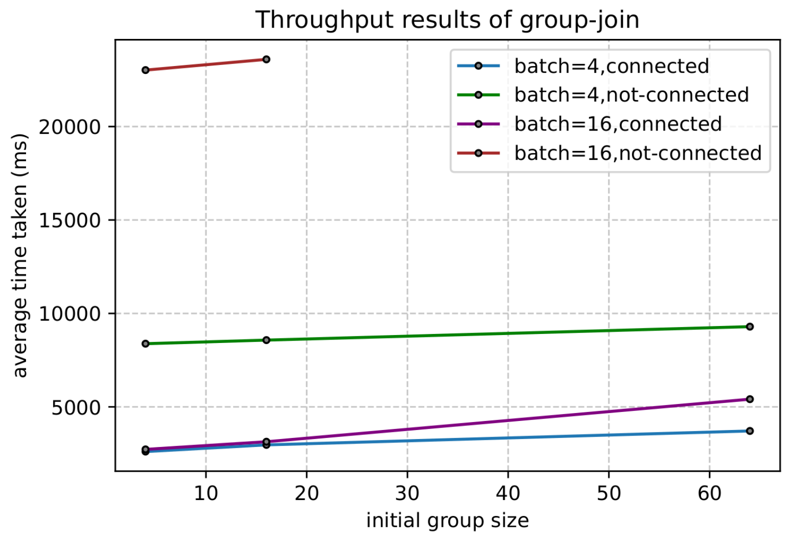


Figure 25: Throughput graph of join

and *thread-safe data structures* can be regarded as one such factor since the implementation includes critical sections where local stores should be updated accordingly. Further, messages can be queued up in both ZeroMQ and internal channels due to the fact that ZeroMQ sockets are not thread-safe and hence messages are processed by each socket sequentially by a single dedicated thread. Therefore, overall latency for a given test-batch constraint can be expected to increase more or less linearly with a slight gradient along with initial group sizes. In addition, the slope of the *not-connected* line can also be anticipated to be higher than a *connected* line, as the *connecting* phase will also have an impact due to the rise in DID-exchange messages transmitted corresponding to the adjustments in group sizes.

The *connecting* phase has even more influence on the latency when a *connected* test case is compared against its identical *not-connected* scenario, due to the requirement of additional $(n-1)*2m$ number of messages where n is the initial group size and m is the number of concurrent joiners. This difference between two identical graph lines can be expected to increase even further with the test-batch size as m in this case is not a constant. For example, {*batch*=4, *not-connected*, *group-size=4*} test case only requires 24 additional messages when compared to its identical *connected* scenario whereas {*batch*=16, *not-connected*, *group-size=4*} requires 96 additional messages over its *connected* test-case.

6.3.2 Group messaging

(a) Performance

This experiment focuses on evaluating the performance of ZeroComm in terms of publishing DIDComm group messages. Each relevant group member in this experiment is registered for a callback such that it notifies the tester once it receives the exact number of a particular message as defined in the registration. Tester carries out these registrations prior to the experiment, then publishes a test batch of messages concurrently and finally measures the time taken until all registered callbacks are received. In order to capture the transport failures, we also defined a timeout such that a member will return the number of successfully retrieved messages in case of a failure. Hence it should be noted that the time measurements of this experiment may also include the cost of parsing an HTTP response body and processing the content, which can be regarded as a trivial overhead against the network latency. However, average latency values were calculated by deducting the ping latency measured for a half round-trip at each test attempt, thus eliminating the impact of callback responses. As both pub-sub and callback connections were already established in advance, any initial overhead related to the transport handshake and routing procedures were excluded in this experiment, thus resulting in latency values slightly lower than in the ping experiment.

In contrast to the previous experiments, an additional independent variable is defined as *mode*, since our proposed model consists of two different flows (*single* and *multi*) based on the number of queues used for transmission of data messages. Moreover, we only considered cases in which all three attempts were 100% successful for the throughput graph due to the uncertain behaviour of failed test attempts with a possibility of skewed and unreliable results.

As shown in Figure 26, two different modes of the solution result in dissimilar latency values when the rest of the conditions are constant across the two scenarios. The cause for this variation lies within subscribers as the publisher does not incur any distinction between the two modes, except for the impotent difference in namespaces of the topic to which messages are published. As described in section 5.2, messages of a specific group in *single* mode are published to the same queue and subscribers are required to attempt with unpacking each message regardless of the intended recipient. Despite all the advantages, this will prompt an additional overhead in terms of resources and processing latency due to the unnecessary invocations to

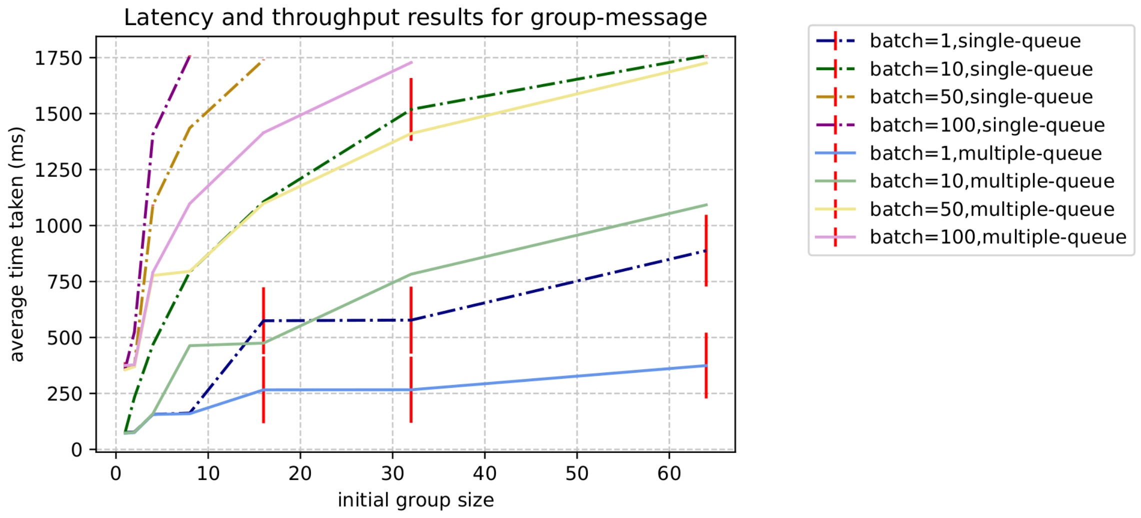


Figure 26: Results of group-message tests

expensive cryptography functions wrapped in *cgo*. In contrast, subscribers in *multi* mode can be expected to be more efficient as they only interact with the relevant messages and do not consume any latency or resources for unnecessary decryptions. Hence, *single* mode latency measurements are anticipated to be higher than the values of equivalent *multi* mode test-cases. This also explains the cause for steeper gradients of *single* mode graph lines as opposed to their counterparts in *multi* mode due to the increase of irrelevant unpacking operations with respect to the number of group members. This overhead consumes even more latency when the number of messages is raised simultaneously since each subscriber is required to unpack *m**(*n*-1) messages in *single* mode where *m* is the number of original messages and *n* is the group size.

Despite messages in this experiment are packed and published concurrently, a number of critical sections and latency overheads are still involved in the procedure. This includes interfering with synchronized in-memory *maps* related to group, key or subscriber, buffering latency in internal *channels*, *cgo* wrapped cryptography functions, usage of a single-threaded publisher socket and underlying *polling* mechanisms within the ZeroMQ context. Thus, latency can be expected to rise also at the publisher side when either batch-size or group-size is increased.

It is worthwhile to note that all the graph lines in this experiment intersect at two different points when the initial group size is one. In order to further investigate this behaviour, an additional experiment was carried out for the specific case of *group-size*=1 using granular message batch sizes as the only independent variable. *Single* mode was used as a constraint since all the messages in this scenario will be only intended for one member and thus the number of invocations to the unpacking process will be consistent regardless of the mode.

Figures 27 and 28 manifest that latency values are composed of step-wise increments with respect to the number of messages published to a single-member group. Further, all these increments belong to the range of 110ms-140ms thus indicating a periodic behaviour of the entire operation. One possible cause for this observation is *Nagle's algorithm* which is often implemented by TCP-oriented applications in order to improve efficiency over network resources by aggregating data bounded by an MTU (Maximum Transmission Unit) size and reducing the number of packets sent over a TCP channel. However, Nagle's algorithm is disabled by default within ZeroMQ transport layer in order to eliminate unnecessary idle times and reduce latency overheads [4].

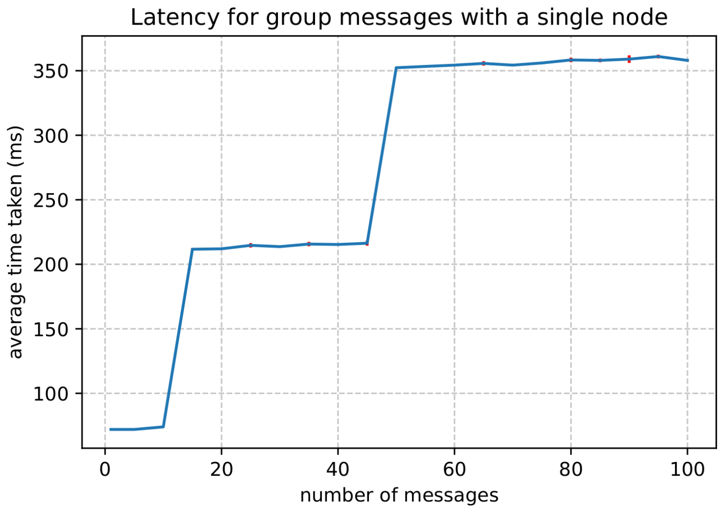


Figure 27: Latency of group-message when size=1, batch=(0,100]

Nevertheless, the transmission of ZeroMQ message frames seems to involve a data batching process at a higher level in which an intelligent queuing mechanism is carried out depending on the data flow and Network Card Interface's (NIC) performance [37]. This implies that if the internal network buffer is sparse, ZeroMQ does not proceed with batching but dequeues messages until it is empty. However if otherwise, background IO threads of ZeroMQ dequeue the buffers for accumulation of data into a batch bounded by the configuration *ZMQ_OUT_BATCH_SIZE*, in contrast to Nagle's algorithm where it waits until acknowledgements are received within the network layer [38]. Thus, the continuous stream of data fed in our experiment enables this

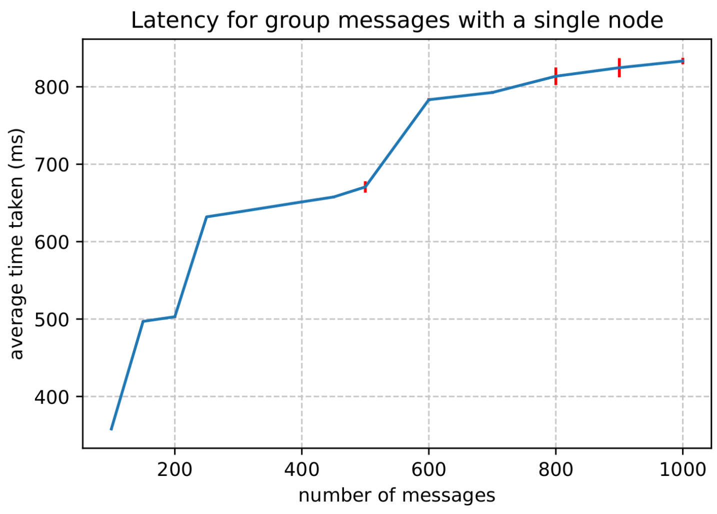


Figure 28: Latency of group-message when size=1, batch=(100,1000]

specific batching algorithm and hence the latency values can be expected to be in step-wise increments. In addition to the efficiency in network overheads, this process also reduces the number of traversals through the entire stack of "Application-ZeroMQ-TCP-IP-NIC" per each message as shown in Figure 29 [39].

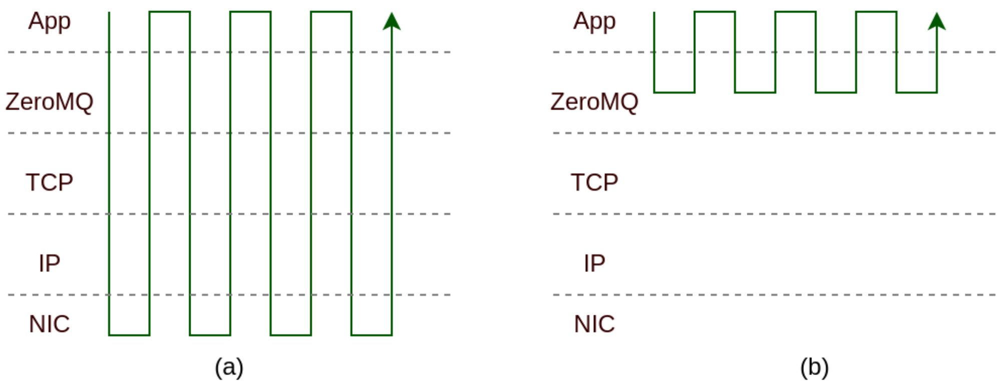


Figure 29: Stack traversals per each message (a) without batching (b) with batching

(b) Efficacy

ZeroMQ is an unreliable messaging protocol that focuses on high throughput and hence follows an *all-or-none* delivery mechanism in terms of multi-part messages. We encountered this symptom during the group-message experiment, specifically when both batch and group sizes were high. Therefore, additional metrics were collected such as successful operations in order to demonstrate how this unreliable delivery of messages affects our proposed solution.

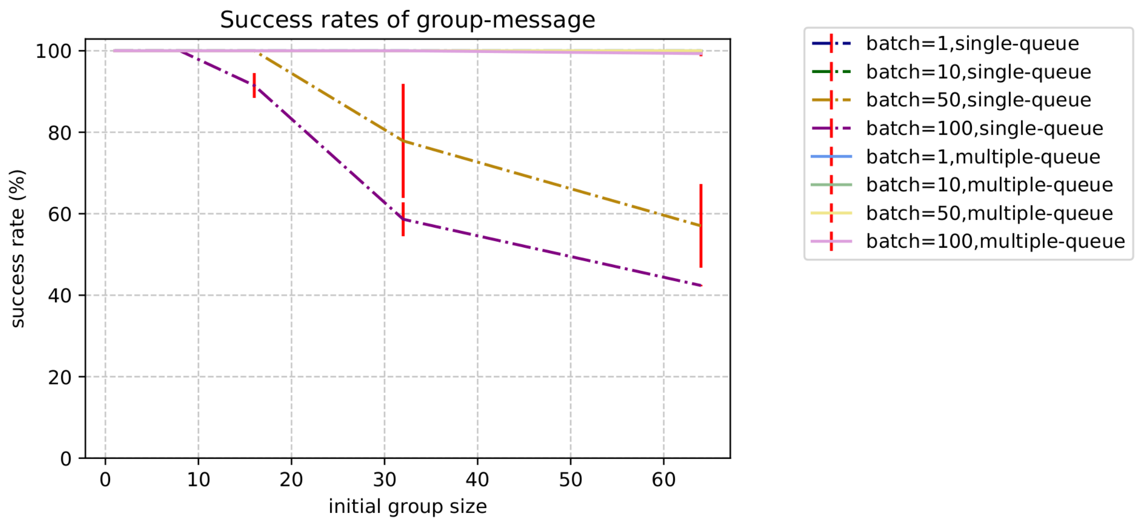


Figure 30: Success rates of group-message

Figure 30 depicts that *multi* mode of our solution produced 100% message delivery except for the case {*group-size*=64 and *batch*=100} in which the average success rate was 99.31%. In this particular scenario, the tester is publishing 6400 messages concurrently intended for 64 different members. At any given point in time, messages can reside in subscriber's or publisher's network buffers, subscriber's memory or even on the wire itself. In order to handle overflowing of ZeroMQ message queues due to such reasons, a configurable parameter is introduced within ZeroMQ parlor known as *High-water mark* (HWM). If this quota is reached by a particular internal pipe, any further message will be blocked or dropped depending on the socket type. Since our publisher socket used the default HWM value which is set to 1000, some messages can be expected to drop for cases with intensive streams of messages.

In cases of *single* mode, a considerable portion of the overall latency is consumed by subscribers for unpacking a substantial number of messages unnecessarily, thus leading to the defect of *slow subscribers*. This potentially causes the publisher to drop messages beyond HWM as subscriber's inefficiency is passed upstream to the internal buffers of the corresponding publisher. Further, TCP channels can also be expected to be overwhelmed with a higher number of network packets caused by the oversized state messages with composed sub-sections. In addition, processing latency by ZeroMQ can vary depending on the message size (as described in section 7.4.1) and this can also impact our solution specifically when state-messages are sizeable. Hence success-rates can be expected to drop immensely in *single* mode even for relatively smaller group sizes when the number of messages published is increased (Figure 30).

(c) Message size

As security is provided at the data layer, messages conveyed via DIDComm can generally be anticipated to be larger in size. However, we conducted separate experiment in order to provide a statistical insight into the expense of ZeroComm in terms of memory occupied by individual group messages.

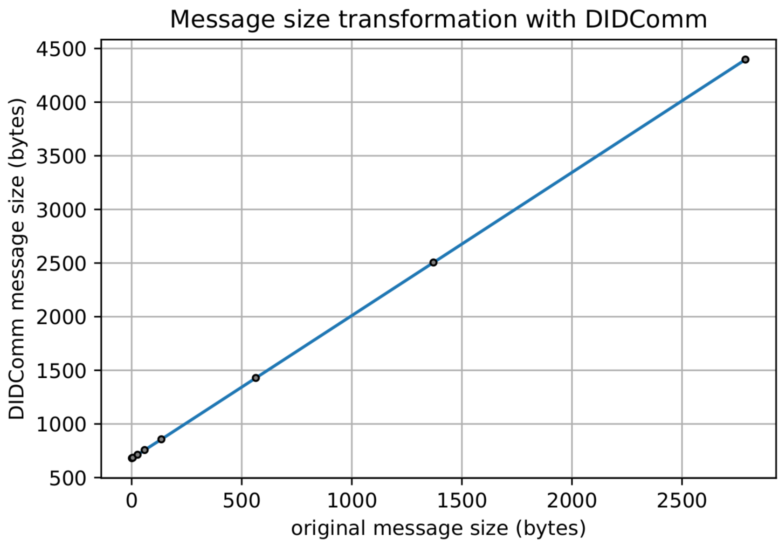


Figure 31: Message sizes with DIDComm

As shown in Figure 31, DIDComm message sizes begin with a fixed overhead even when the original content is only a single byte, thus minimizing any correlation between message size and content. This is caused by the use of a predefined structure in DIDComm regardless of the original text as well as by the encryption algorithms enforced on the primary content. It can also be observed that resultant DIDComm sizes exhibit a linear relationship with the initial size of the message. This can be argued by the increase in size of the *ciphertext* field (Figure 9) with respect to the original message, whereas the rest of DIDComm message attributes are not impacted by the initial size of the content.

During the design phase, we assumed that state changes are relatively infrequent compared to the data messages. Hence, we used a single status message with subsections which comprise the identical message but are packed per each member in accordance with DIDComm protocols. This however poses a notifiable hindrance in terms of the message size as it increases with the cardinality of members. Specifically, this growth can be expected to be linear since the number of subsections is directly proportional to the group size while each segment results

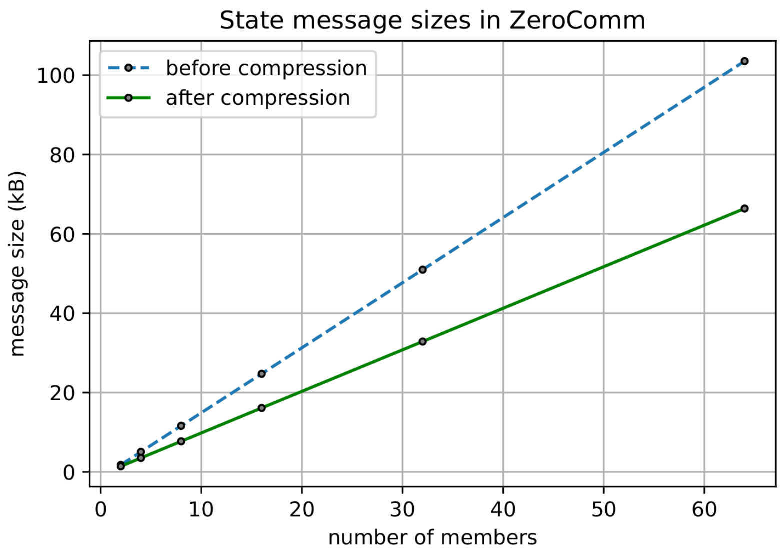


Figure 32: State messages in ZeroComm

in an approximately equal number of bytes (Figure 32). This impact is minimized with *zstd* compression algorithm used in our solution but still, this can be regarded as a potential future work to further improve state propagation with respect to the alternatives mentioned in section 4.4.

7 Discussion

This chapter reflects on the qualitative attributes of ZeroComm while investigating the internals and design choices. Accordingly, we dive into the specifics of resultant trust and security, prevalence of multiple group views, ordering of data messages and key features of messaging at both transport and protocol levels.

7.1 Trust and Security

7.1.1 End-to-end encryption

The main objective of our study is to incorporate desirable safety properties of communication into the core model of pub-sub messaging pattern via exploiting DIDComm protocols. Despite the fact that current pub-sub based systems exhibit adequate security levels through sophisticated cryptography measures, they still fail to satisfy the requirements of an ideally trustful environment due to the involvement of an external component. For example, Internet Relay Chat (IRC) serves as a decentralized group communication model but however, it still poses a vulnerability with security since messages are not end-to-end encrypted by default and even if it is enabled by add-ons, messages will be encrypted by the server and not by the client [40]. In contrast, our solution eliminates external components by using ZeroMQ-based individual agents and entails security at the data-level via DIDComm protocols thus preserving both decentralization and end-to-end encryption.

7.1.2 Transport authentication

Once DIDComm connections are successfully established, every single message used in all subsequent processes (e.g. joining groups, updating member information, peer-to-peer and group communication) adheres to DIDComm protocols. This does not only provide confidentiality via encryption but also the authenticity which boils down to the initial DID-Exchange between the two peers. For instance, encrypting sender's verification key with receiver's public key in a message, allows the intended recipient to correlate a message with the sender. However, these securities only exist in the data layer of the stack.

ZeroMQ's inter-node communication is based on TCP connections and their corresponding *PUB* sockets allow any node to connect with the port to which it is bound, thus creating a communication channel as long as the requested socket type is compatible with *PUB* (Figure 33). Despite that content of messages is protected with DIDComm and endpoints are only conveyed to the validated DIDComm agents, this still poses a threat with obfuscating network traffic. Particularly, if the endpoint is somehow exposed, a malicious node can then connect and monitor the messages continuously along with any pattern of messaging involved. To this end, topic names can be generated such that they are unique and known to other group members only via DIDComm, thus restraining disclosure of messages by resolving a transport-level defect at the group protocol layer.

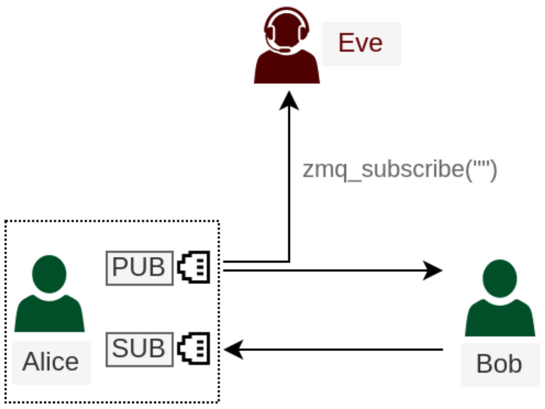


Figure 33: Intruder connecting via ZeroMQ sockets

However, ZeroMQ allows subscribing to all topics via an empty string (“”), thereby enabling unknown subscribers to read all the messages without any interference from the publisher [41]. This implies that the issue should be resolved at the transport layer itself rather than by a higher-level protocol and hence we used ZeroMQ's inbuilt *authenticator* in our solution [42]. Although it provides several options such as filtering IP addresses, username-password validation and curve-based authentication, the latter was chosen for our solution due to security being the highest priority among requirements. Specifically, this authentication uses *curve25519* elliptic curve for generating permanent and transient (for each session) key pairs to authenticate both publisher and subscriber as well as to enforce *Perfect Forward Secrecy* [43].

Nevertheless, using an authenticator still causes a complication due to the absence of a ZeroMQ key-exchange mechanism which requires the generated permanent key pairs to be transferred manually. To this end, we used the high-level subscribe message of a member to securely communicate the ZeroMQ key-pairs via DIDComm, thus eliminating both the necessity of additional messages and the vulnerability of an external key-exchange. In essence, ZeroComm only uses a single initial out-of-band message and all the subsequent processes are securely achieved via DIDComm, including group memberships and transport authentication.

7.1.3 Network security

A number of vulnerabilities can still exist in a communication system as given below, despite that messages are transmitted with end-to-end encryption.

- **Session information:** *Single* mode in our solution minimizes network obfuscation via publishing all group messages to a single topic. Nevertheless, the mere presence or absence of traffic may still be used by malicious parties to collect information regarding the sessions. To this end, agents can be implemented to send random *garbage data* when there is no traffic among the group members.
- **Versioning:** Our solution uses a curve authentication which is provided by ZeroMQ v4.0 implementation. The initial handshake of two ZeroMQ peers includes a version negotiation in their *greeting* messages which results in a vulnerability of disabling the enforced security mechanism with a lower version. However, the corresponding ZMTP implementation used in ZeroComm eliminates any attempt with such *downgrade attacks* [30].
- **Message size correlation:** Attackers use the length of messages to extract insights about the content even if they are encrypted. As we described in section 6.3.2, DIDComm messages in our solution exhibit a linear relationship with the original size thus providing a susceptible correlation. Nonetheless, this can be resolved by *padding* the encrypted data in each message to a randomized size.
- **Repeated requests:** If service endpoints of a ZeroComm agent is somehow exposed, an attacker may continuously endeavour to connect with the agent such as in DDoS (Distributed Denial of Service) attacks [44]. This issue can be minimized by maintaining an *access log* in order to detect and block repeatedly failed connections that have been originated from the same IP address.
- **Oversized messages:** As DIDComm protocols result in sizeable messages, a malicious agent can attempt *amplification attacks* by sending requests to generate large responses

in return. This should be handled by the application layer depending on the use-case as our study only focuses on enabling secure group communication.

7.1.4 Key management

The essential security aspects of our solution are delivered through the use of symmetric and asymmetric cryptography key pairs. Hence proper measures should be considered for the construction of a well-established Public-Key Infrastructure (PKI) which may result in malevolent consequences if not carefully designed [45]. To this end, ZeroComm adheres to a set of design choices but however, a number of assumptions were also made to maintain simplicity and give prominence to the core algorithm of the solution as described below.

- Separate key pairs are used in invitation and DID-exchange protocols even if they are related to the same recipient. This minimizes the correlation between two protocols, enhances security and proves authenticity during the handshake and subsequent messages. However, our model does not provide any assurance on the out-of-band transmission of the invitation as it is out-of-scope for this study.
- Same key pair is used for all the invitations generated by a single member to lessen the burden of complexity in our solution. But ideal implementations should use a new key-pair for each invitation such that it decouples the relationships maintained with distinctive users by a particular agent.
- All key pairs used in an agent are stored only in memory (no persistence) and as a slice of bytes (not human-readable) for both better security and efficiency. In addition, key-manager service of ZeroComm is separated and abstracted such that further mechanisms can be applied in more practical scenarios. As an example, peer public keys and agent's own key pairs can be stored separately to decouple and enforce granular security such as splitting agent's keys when not in-use.
- A single key pair is used by an agent in our proof-of-concept implementation for all the services and topics maintained with a specific peer which is identified uniquely by a label (eg: Alice uses only K_{pub}^{AB} and K_{prv}^{AB} key-pair with Bob regardless of the service). Nevertheless, multiple key pairs should be used for each distinctive use-case even within the same relationship to achieve better privacy.

- As described in section 7.1.2, additional keys are required to enable transport-level authentication and exchange of this underlying key-pair is accomplished through the subscribe message in ZeroComm.

It should also be noted that due to the evolution of quantum computers, the existing systems including our solution may inherit vulnerabilities with unraveling protected messages, commonly known as ”store now decrypt later” attacks. As an example, 20 qubits can represent over a million superimposed states thus leading to at least one-million simultaneous computations including cracking public key pairs with Shor’s algorithm [46]. However, quantum-resistant key generation mechanisms are currently being researched by the community specifically with lattice-based algorithms which still preserve the difficulty of deciphering keys regardless of the computational power [47]. Due to the abstraction of key manager and packing interfaces in our solution, corresponding cryptography functions can therefore be replaced as necessary without impacting the core functionalities, thus making the entire model quantum-secure.

7.2 Group views

Due to the replication and maintenance of state by individual members, inconsistent views may exist within the same group which can be caused either by network delays or byzantine nodes. The latter may possibly result in security vulnerabilities as it enables an existing member to include an intruder in its local view and subsequently share it with newcomers. This however can be resolved by enforcing consensus with *Virtual Synchrony* into our solution (eg: Paxos) such that it does not only maintain a consistent view across a process group but also provides atomic multicasting mechanisms [48]. However, we presume that this approach may result in unnecessary complications and overhead in our solution, since we primarily focus on providing secure group communication within DIDComm context. Accordingly, ZeroComm is constructed atop individual DIDComm agents, which implies that trust should prevail explicitly between each pair of agents even though they belong to the same group since technological reliability alone is insufficient to provide communication safety as described in section 2.1.

Nevertheless, we investigated how inconsistent group views may influence our solution along with an approach to mitigate such adverse impacts. In particular, the join algorithm of our solution is susceptible to byzantine nodes since a newcomer connected to a malicious acceptor may possibly receive an incorrect group view, thus joiner initiating imprudent connections with members outside the group. Given below is a list of example scenarios that may have an impact by the existence of such different group views.

Scenario 1: Alice creates a group. Bob sends a join request to Alice. Alice sends group-state with an intruder (Eve).

Scenario 2: Alice and Bob are members with inconsistent views such that Alice has an intruder (Eve) in her local view. Carol then sends a join request to Alice (Figure 34).

Scenario 3: Alice and Bob are members with inconsistent views such that Alice has an intruder (Eve) in her local view. Carol then sends a join request to Bob.

Scenario 4: Alice and Bob are members with valid inconsistent views. Carol sends a join request to Alice.

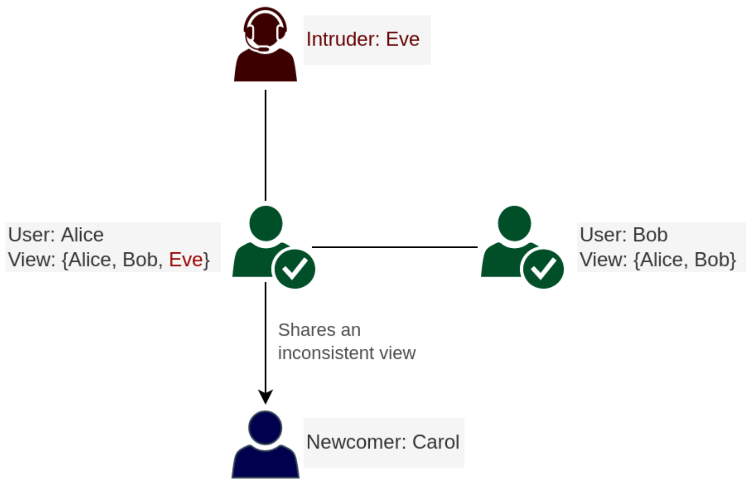


Figure 34: Join process via a byzantine acceptor

Alice acts as a byzantine member in the first two scenarios, leading newcomers to connect with an intruder and even include Eve in their local views of the group. Since scenario 1 should ideally consist of only one member (Alice), it will be straightforward for the second member (Bob) to reject connecting with any other member than Alice. However, scenario 2 results in a vulnerability as Carol may likely connect with Eve in cases where Carol is unaware of the actual group size. In the third scenario, Alice does not act as an acceptor and hence her flawed local view of the group will not be shared with Carol. Since both Alice and Bob are valid members with only inconsistent views in the last scenario, Carol may have an impact by Alice but the root cause in this case is unintentional.

In general, newcomers connecting to a group with multiple members should be capable of validating the initial view provided by an acceptor, with respect to the other members. To this end, every member in our solution maintains a checksum value per each distinctive group based on their connected members. This hash is calculated by the *validator* component (Figure 19) with a consistent algorithm such that the resultant value will be identical given the same set of group members. Each existing member will convey the most recent checksum value to a joiner via subscribe-response during the *connecting phase* (Figure 24), which will then be validated by the joiner once all the individual DIDComm connections are completed. Hence this can be regarded as a reactive approach to recognize byzantine nodes in a group and thus avoids the additional overheads incurred by proactive measures.

Nevertheless, our solution only focuses on recognizing whether each member maintains an identical view of the group and thus ZeroComm does not provide any further rectification on resolving byzantine failures. Distinguishing and sorting out byzantine nodes against valid members with only partial views should therefore be investigated as a separate study. However, a parameter is introduced in our model such that a group can be configured to terminate any join procedure when the group views received by members are incompatible.

7.3 Consistency

The degree of consistency can be regarded as a highly concerning factor in our solution due to the inherent asynchronous behaviour of a distributed group. Particularly, the transmission of messages in parallel by multiple members may lead to inconsistent ordering of messages across different members. This section provides a review of how consistency can be enforced in ZeroComm by considering such possible scenarios and constructing a model based on the alternatives.

1. As the most naive approach, ordering of messages can be maintained via a **centralized** component, thus requiring a message to be submitted initially for the ordering process and then published to the group with a defined sequential index. Despite the simplicity, this consumes a higher latency overhead due to the synchronization and additional

round-trip of each message. Moreover, the model may possibly suffer from *single point of failure* and a network bottleneck with respect to the centralized component. The coordinating node can be replicated to overcome this issue but it results in further complications such as management and load-balancing of the resultant replicas. In addition, this centralized approach also contradicts our primary design goal of decentralization and hence can be justifiably discarded as infeasible for ZeroComm.

2. As an alternative, ordering of messages can be achieved via **totally ordered multicasting**, thus entirely eliminating the need for a centralized component. However, this approach can possibly lead to even higher latency overhead and network congestion due to the involvement of an additional multicasting round, multiple acknowledgements, processing with more queues and a waiting time bound with each message. As a result, we omitted this strategy in our solution but still can be considered in future extensions of ZeroComm, specifically where ordering is favoured at the expense of performance and complexity.

3. In practice, not every application requires to be strictly ordered but rather settles for **hybrid models** with respect to data integrity and constraints of the system, thus minimizing unnecessary complications and latency overheads [49]. We explored a number of scenarios where consistency may possibly impact the solution and constructed a hybrid model accordingly with partial ordering based on the assumptions as described below.
    (a) Messages published by a single member should be strictly ordered across the entire group via **sequential consistency** and should preserve **monotonic-writes** with respect to the client-centric consistency model (however, it should be noted that client-server paradigm only exists as momentary roles in our model given a particular message transmission). For example, if member *A* publishes messages {m_1, m_2, m_3, ...} in group *G*, all the other members of *G* should read *A*'s messages in the exact same order (Figure 35). This requirement is satisfied through the FIFO property of message-queues used in the core of our solution.
    However, it often results in complications with ordering messages when multiple publishers are involved, which can be resolved in several ways.
    (b) Since it is technically infeasible to maintain an absolute global time in Wide Area Networks, ordering among concurrent events incurs additional overheads and com-

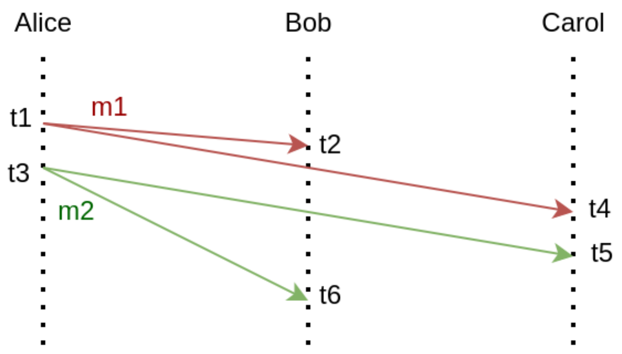


Figure 35: Message ordering by a single publisher

plexity. In cases where they are causally unrelated, we assumed that the messages are parallel and independent events that do not require to be ordered. For instance, Figure 36 depicts a scenario where Bob receives m_1 after m_2 despite that m_1 was in fact published ahead of m_2. This also implies that our solution may not be applicable to all the use-cases, specifically where strong consistency is mandatory such as financial applications which use monetary transactions as messages.

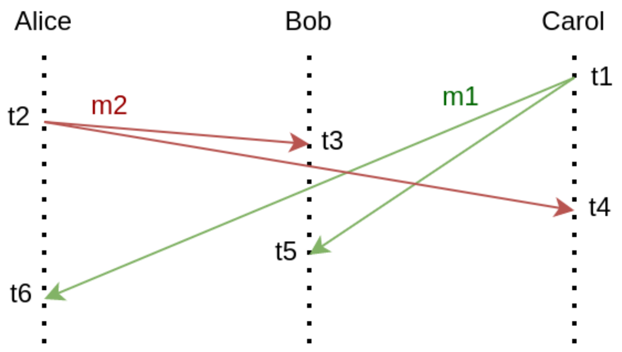


Figure 36: Ordering with multiple publishers

(c) When messages are causally related, granular mechanisms can be considered and applied since our model involves two types of messages. In particular, ordering of state messages can be segregated from data, leading to the assumption that causally related but context-wise unrelated messages can still be in different orders and thus do not necessitate any ordering mechanism.
As an example, Figure 37 shows a case where Alice receives m_1 before join-status of Carol even though it was published prior to m_1. Consequently, Alice publishes m_2 only to Bob despite that it should ideally be transmitted to Carol as well, due

to Alice being unaware of Carol's existence at the time of dispatch. This however can be improved by attaching a notion of the group state in each message, thus even contextually unrelated but causally related messages provide insights to other members on publishing a future message. In our given example, m_1 thus may carry information including that either a member has joined or group-state has been updated since it is causally related to the join-status of Carol. Upon retrieval of m_1, Alice can then proceed with a reaction such as waiting until the recent group-state is restored in its local view. Nevertheless, this mechanism may become overly complicated in the presence of multiple ephemeral members and hence assumed that state can be disseminated eventually throughout the group. Consequently, this restricts ZeroComm to use-cases where delayed propagation of memberships can be tolerated.

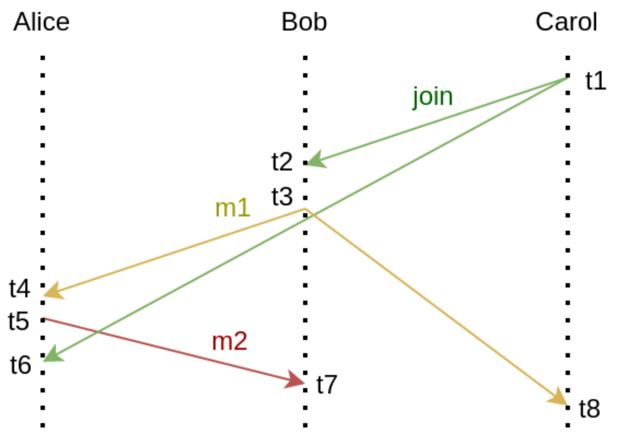


Figure 37: Ordering among different contexts

(d) In contrast, causal order should be preserved if messages are causally as well as contextually related such that no data message published as a response of another will be received ahead of its predecessor as shown in Figure 38. However, it should also be noted that m_2 and m_3 in this example are received by Carol in a different order as they are not causally related to each other.

In essence, our hybrid model will be based on both parallelism and type of the message in which states will be **eventually consistent** whereas data will be **causally consistent** among multiple publishers. In addition, both state and data messages of a single publisher will be **sequentially consistent**. We presume that any rectification on consistency (eg: re-ordering

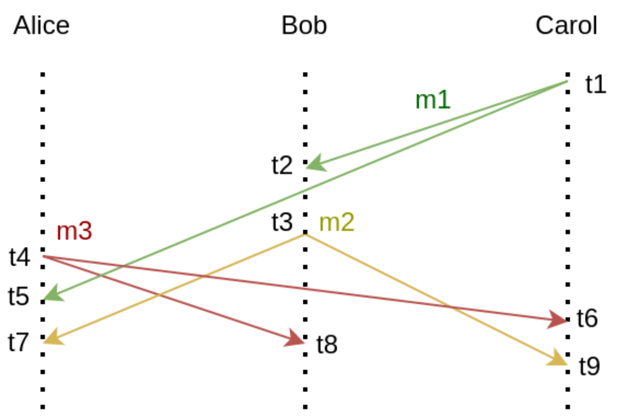


Figure 38: Causally ordered data messages

messages) should be implemented in higher layers of the stack depending on the use-case (i.e. application-layer in Figure 16), without impacting the core algorithm of the ZeroComm model. However, lower layers of the model should facilitate any underlying framework required by applications in order to maintain such consistency levels. To this end, an additional tuple {*lamport_ts, id*} is attached with each message, where *lamport_ts* is a logical timestamp maintained by each agent with respect to a Lamport clock as follows [50].

```
if current_ts < last_read_ts:
    lamport_ts = last_read_ts + 1
else:
    lamport_ts = current_ts
```

In this case, *current_ts* refers to the present UTC (Universal Time Coordinated) timestamp maintained locally by a member while *last_read_ts* refers to the most recent timestamp attached in a message processed by the same member. Further, an *id* attribute is generated randomly during the bootstrap phase of a publisher which can be used to sort messages in cases where *lamport_ts* values are equal across multiple messages read by a subscriber. Accordingly, this *id* together with *lamport_ts* can be used by an application to execute the ordering of data messages (eg: buffer messages and sort based on the tuple values).

7.4 Messaging

7.4.1 Transport

ZeroMQ plays a vital role in our solution despite that the transport layer is abstracted via interfaces and this section explores a number of design choices specifically related to the internals of ZeroMQ.

1. **ZeroMQ sockets are not thread-safe:** This results in undefined behaviour when sockets are invoked concurrently by event flows. Thus, the internal architecture of our group-agent is entirely based on message-passing such that no socket is shared between multiple threads. To this end, a group-agent uses both inbuilt *go-channels* as well as internal endpoints bound with *inproc* ZeroMQ sockets. The latter enables the same *SUB* socket which is exposed externally for group messages, to be also used for the internal communication since multiple connections are allowed by the underlying ZeroMQ socket regardless of the transport being used.

2. **Missing message syndrome:** ZeroMQ favours throughput over reliability and thus sockets are asynchronously connected with each other. This leads to a possibility of missing the initial messages published by a member even if the *PUB-SUB* socket connections were invoked in advance. This syndrome affects our solution since a new member in ZeroComm publishes its active status to the group as soon as the individual DIDComm connections are established with all the members. We identified a number of possible methods to overcome this issue as listed below.
   (a) As a naive approach, the newcomer can *sleep* before publishing its status, thus allowing underlying socket connections and ZeroMQ handshake to be completed in time. However, this can be regarded as an inefficient mechanism due to the additional latency overhead as well as manual sleep configuration required in each agent based on the network traffic.
   (b) Alternatively, *callbacks* provided by the core library can be used such that it notifies any update on the transport layer including connection establishments. This was also disregarded in our solution due to the lack of adjacent peer information in a callback, which is necessary when multiple peers are involved in our model.

(c) As a more robust solution, a newcomer can publish a stream of *hello* messages prior to the real data and its active status. A subscriber responds back with an acknowledgment which consists of a specific identifier upon the retrieval of this multi-part hello message. The newcomer will publish any subsequent message (eg: active status) only after the acknowledgements are received from all the members since it verifies the successful connection with each group member. Hence we adopted this method in ZeroComm which enables the group protocol to dynamically adapt depending on the network traffic while improving efficiency.

3. **Message size:** Messages are processed differently by ZeroMQ based on the size, where small messages are stored on the stack while large messages are stored on the heap. It minimizes unnecessary memory allocations while improving the overall latency [51]. This specific behaviour can possibly impact our solution in cases where state messages are oversized due to the higher number of members in a group. Message-size constraint is configured by ZMQ_MAX_VSM_SIZE variable which is set to 30 bytes by default but can be overridden as required [52].

4. **Bi-directional flow:** In terms of state messages, every individual relationship in a group can be considered as duplex. To achieve this two-way communication, we used the fundamental socket types *PUB* and *SUB* thus creating two uni-directional data streams. However, the solution can be redesigned to use ZeroMQ's extended socket types instead, such as *XPUB* and *XSUB* which enable bi-directional data flow within a single connection by maintaining a double queue internally for incoming and outgoing messages [53].

5. **Property frame:** As our solution was required to handle messages differently based on the type, we attached a metadata object in each message to convey any additional information. The current implementation uses only a *type* attribute with encoded and implicitly defined integer values such that they are incomprehensible outside the ZeroComm context. We also formulated our multi-part message structure to include these properties in an additional ZeroMQ frame thus preserving the compatibility of ZeroComm with different versions. However, this still poses a vulnerability by partially revealing the context of a message and hence should be eliminated in future implementations such as by encrypting the metadata frame as well.

7.4.2 Protocol

DIDComm serves as the most crucial component in ZeroComm as our key requirements such as trust and security rely entirely on the underlying protocols. Nonetheless, a number of design choices were considered during the implementation phase as described below, in order to diminish any further complication apart from the core algorithm while maintaining sufficient security throughout the process.

1. **Peer DIDs:** DIDComm supports a variety of DID methods (eg: *key*, *ethr*, *btcr*, *github*) each with different properties and storage mechanisms. Nevertheless, *Peer* DIDs are used in our solution resulting in a ledger-agnostic communication model since interoperability is not a primary objective of this study. However, integration of other DID methods can be considered as required by any future implementation of ZeroComm.

2. **Synchronous and duplex messaging:** In order to capture all messaging use-cases and to support ubiquitous devices with unpredictable intervals of stable connections, DIDComm recommends the use of asynchronous and simplex communication [23]. This does not only provide obfuscation but also interoperability as any advanced communication system can be constructed atop this basic messaging pattern. However, peer-to-peer messages involved in our group-join protocol are synchronous and duplex such that responses are transmitted through ZeroMQ *REP* sockets for the corresponding requests, thus leading to a simpler and more efficient model.

3. **Service endpoints:** A DIDComm agent conveys its functionalities through service endpoints of a DIDDoc. Each service ideally consists of a separate key pair and a network port to provide granular security and decoupling between multiple endpoints. However, these attributes were implemented to be identical across the services of a single agent in this proof-of-concept model.

4. **Unique and consistent labels:** In practice, agents may use different labels for independent peers, groups as well as service endpoints to achieve better privacy through minimal cross-references. Conversely, our prototype naively assumes that labels are unique and consistent across each scenario.

5. **Invitations:** Distinctive invitations should preferably be used in setting up DIDComm connections to maintain both traceability and decoupling between relationships. However, the same invitation is used by a group member to join with newcomers in ZeroComm for simplicity reasons.

6. **Secure group protocol:** The proposed group communication model operates on top of the DIDComm layer. In particular, each message between a pair of agents is transmitted only via DIDComm preceded by a secure DID-Exchange procedure. This implies that all the messages used in our group protocol (eg: requesting information, subscribing a member, leaving the group) can be regarded as secure and trustful.

7. **Idempotent state transactions:** Agents in our solution were implemented to cope with state-message duplicates thus allowing members to republish a status in case of a failure.

7.5 Application Layer

ZeroComm can be expected to provide the fundamental structure of communication for numerous use-cases and it is worthwhile to explore the model from the perspective of an application. To this end, a group-chat service can be regarded as an ideal use-case since we adopted a hybrid model with assumptions to loosen consistency levels while prioritizing security and trust of individual messages. Hence, this section provides a detailed analysis of our solution with respect to a de facto Instant Messaging (IM) model.

- Our model is a **synchronous** system as opposed to generic chat applications. For example, *Whatsapp* is an asynchronous system where intermediate servers store and deliver messages later if the recipient is not online at the time of transmission. It is possible for a member in our solution to stay offline by publishing an inactive status but however, any published group-message during this inactive period can not be retrieved by the member.

- As the solution is based on the ToIP framework, trust is embedded at different layers of the network stack, including messages themselves. Hence **secure-routing** can be considered as non-compulsory compared to Internet Relay Chat (IRC) where messages are transmitted in plain-text by default. Nevertheless, *mediators* in DIDComm can be used to further enhance network security by encrypting a message with multiple envelopes per each intermediary as similar to *Onion Routing* [54, 55]. Additionally, DIDComm mediator nodes can autonomously enforce rewrapping and delayed transmission as described in section 2.3.

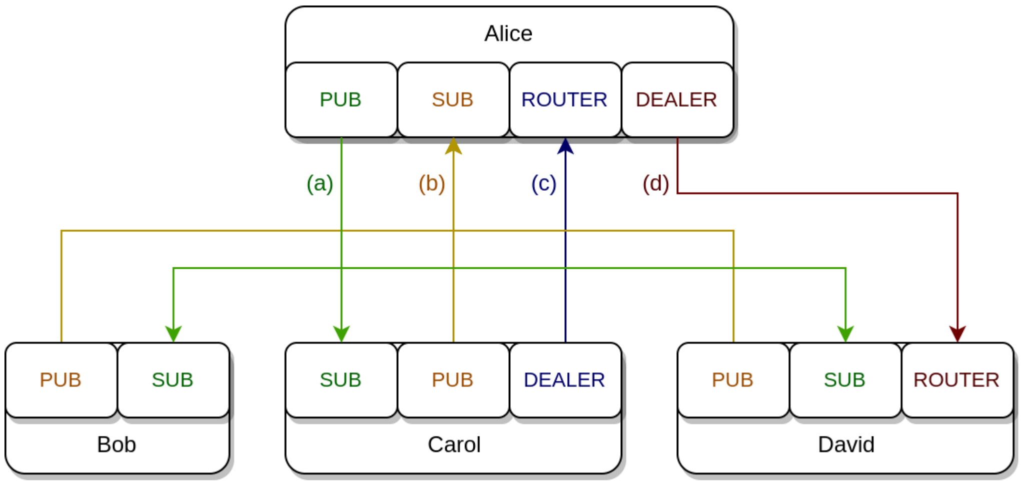
- **Acknowledgements** of messages are ignored in the current implementation as this only serves as a Proof-of-Concept. However, they should be considered in real-world scenarios and encrypted as per DIDComm protocols for a reliable and secure messaging system. It should also be noted that sending *acks* in plain-text often results in undesirable consequences, as experienced with existing IM applications [56].

- In contrast to the conventional chat systems, decentralization is preserved throughout our entire solution. This implies that the model should maintain a flat hierarchical structure within the group in terms of **sovereignty**. To this end, admin privileges including the removal of a group member were neglected in ZeroComm and therefore, a consensus algorithm should be integrated with the solution in order to facilitate removing members, specifically in case of a byzantine failure.

- In this Proof-of-Concept, we only considered text messages to maintain the simplicity of the solution despite that existing IM applications support Rich Communication Services (RCS) by default. Nevertheless, messages in our solution can be easily modified to include DIDComm *attachments* in order to transmit **multimedia messages** with arbitrary data formats such as images, videos and documents [57].

- ZeroComm does not currently support **referencing** to a previously published message. However, it can be achieved in our solution by maintaining a unique ID attribute in each single message and storing both content as well as ID securely in the application layer for any future reference.

- Retention of messages can also be implemented in the application layer in order to deliver **message history**, such that it preserves data security both when in motion and at rest. In particular, the implementation should maintain and transmit snapshots of data via *ROUTER* and *DEALER* sockets established in publisher and subscriber respectively. Consequently, when a subscriber is initialized or recovered after a failure, it should request for the snapshot while buffering current data messages in the underlying *SUB* queue. Once the response is received and message history is restored, all the buffered messages up to the snapshot can be discarded while the rest should be processed accordingly.



(a) Alice publishes her messages to the group
(b) Alice receives messages from other members
(c) Carol requests message history from Alice
(d) Alice requests message history from David

Figure 39: Restoration of message history

For efficient usage of resources, every member does not necessarily need to provide *ROUTER* sockets or maintain message history. Instead, few members can be chosen to deliver this service with a desirable replication factor to avoid single point of failure (Figure 39). This set of members should be flagged and recognizable in the initial state-response shared during the join process of a newcomer.

- Existing messaging applications including TLS 1.3 implementations, use *Perfect Forward Secrecy* (PFS) to maintain session-specific keys and preserve the **confidentiality of historical communication** in cases where session keys are compromised. *Signal* achieves this via *Double Ratchet Algorithm* while *Whatsapp* only exploits the *Symmetric Ratcheting* process [56]. However, DIDComm perceives PFS from a different viewpoint which is currently accomplished through the rotation of DIDs themselves as the definition of a session is slightly altered in the DIDComm context. Thus, DID rotation should also be considered in a real-world use-case of our solution.

- As ZeroComm provides two dynamic roles based on the member's preference (*read-only* or *read-write*), transitions between these roles should also be feasible. Currently, this can only be achieved by leaving a group and rejoining with the preferred role again since **switching between roles** is out of the scope for this study. Hence, any future implementation can include this functionality as well in order to enhance the overall user experience.

8 Conclusion

This study was focused on the successive step of technological communication known as web 3.0 which prioritizes decentralization, proper data ownership, ubiquitous connectivity and self-sovereign identity. We identified that DIDComm delivers secure and trustful communication in this paradigm which still needed to be explored with group messaging, as all the prior work of DIDComm was only restricted to peer-to-peer models.

To this end, we investigated the requirements of a decentralized and safe group communication model along with its key features, alternatives and trade-offs. Accordingly, we designed and implemented ZeroComm using publisher-subscriber pattern of ZeroMQ while embodying trust and security in messages via DIDComm. We also introduced two different modes of ZeroComm which were then experimented to evaluate the performance trade-off carried out during the design phase against security and decentralization. Section 8.1 includes the concluding remarks of our study followed by a brief description of future work related to ZeroComm.

8.1 Remarks

We identified that combining group communication with decentralization and end-to-end encryption leads to conflicting conditions as opposed to the generic bi-party models. When defining the system specifications, we traded off performance and efficiency for higher security by exploiting hardware and network resources with supplementary cryptography computations, demultiplexed DIDComm messages and granular pub-sub relationships among the members. ZeroComm copes with dynamic group memberships by decoupling state messages from regular data and enforcing different mechanisms based on the message type but still utilizing the publisher-subscriber pattern. As DIDComm entails security at the data layer, sizes of individual messages involved with ZeroComm can be considered to be relatively high. This specifically becomes noteworthy for state updates in large groups but can be overcome with alternatives as we investigated in this study. Our model also incurs additional entry and exit costs per each group member with DID-Exchange protocol and graceful shutdown, respectively.

In essence, ZeroComm serves as a group-messaging protocol that is constructed atop DIDComm peer-to-peer layer by only preserving and extending its desirable properties. This implies that all the subsequent processes involved in ZeroComm are accomplished through DIDComm messages. The join algorithm of a member can be regarded as the crux of our solution in which security and trust are derived from the underlying DIDComm protocols. ZeroComm provides a mechanism to recognize potential byzantine members but recovering from such failures should be handled separately. In addition, our proposed model supports a hybrid consistency model for group messages but any imposition with ordering messages should be implemented in the application layer.

Further, ZeroComm offers two varieties of achieving group communication based on the organization of message-queues among the participants. Specifically, *single* mode in which all group messages are published to the same topic, offers higher security through obfuscating network traffic and simplicity via using the default overlay network constructed by ZeroMQ. Nevertheless, this approach produces low reliability and high latency overhead in cases where both group size and number of messages are sizeable. In contrast, *Multi* mode maintains distinctive relationships among members within the same group and thus under-utilizes the pub-sub pattern as messages are decoupled with a dedicated message-queue per each relationship. This method however provides reliable delivery of messages with lower latency and processing overhead.

8.2 Future work

A number of alternative design choices were considered in the course of constructing ZeroComm model in order to satisfy the defined set of system requirements. In each design choice, we prioritized security over performance and therefore, alterations in such cases can result in different flavours of our solution, which may be more applicable in certain scenarios. For example, the state of a member can naively be propagated via a shared key mechanism thus improving performance and resource utilization at the expense of security. This can even extend our comparison of two modes (*single* and *multi*) by experimenting with other possible varieties of ZeroComm corresponding to the different combinations of design choices.

This study primarily focused on group communication despite that ZeroComm uses a novel transport protocol for peer-to-peer communication. Therefore, the current implementation provides the potential for a separate study regarding how ZeroMQ performs as a transport with respect to individual agents. Similarly, the proposed model for group functionality can be evaluated with other pub-sub frameworks apart from ZeroMQ, by integrating the corresponding libraries via abstracted interfaces but still preserving the decentralization.

ZeroComm only offers the backbone of a communication model and hence application-oriented features were disregarded in its core algorithm. This leads to the possibility of extending the implementation to fit better with specific service-level requirements. For instance, the adopted consistency model can be changed accordingly depending on the degree of message-ordering required. Moreover, granular mechanisms to cope with byzantine members can be implemented even though we provided a configuration to restrict joining groups with inconsistent views. ZeroComm can further be extended with more specific requirements such as supporting late joiners by restoring message history, asynchronous group members, crash recovery, reliable delivery with slow subscribers and perfect forward secrecy.

On a final note, the current implementation serves only as a Proof-of-Concept and should not be used in production without any rectification. As an example, we used a dummy validation in the acceptor to approve a newcomer during the join process, prior to sharing the corresponding group information. The exclusion of message acknowledgements in our prototype can be regarded as another instance that should be considered in a real-world application. In addition, interoperability of ZeroComm must be assessed using community standards due to the abundance of agents developed by multiple vendors on DIDComm which will be ultimately in need of communicating with each other.

References

[1] Joseph V. DeMarco and Brian A Fox, "Data rights and data wrongs: Civil litigation and the new privacy norms," *The Yale Law Journal*, vol. 128, pp. 1016–1028, April 2019.

[2] D. M. Dominic Rushe, "Zuckerberg sued by dc attorney general over cambridge analytica data scandal." `https://www.theguardian.com/technology/2022/may/23/mark-zuckerberg-sued-dc-attorney-general-cambridge-analytica-data-scandal`.

[3] "Introduction to Trust Over IP," White paper 2.0, Trust Over IP Foundation, November 2021. `https://trustoverip.org/wp-content/uploads/Introduction-to-ToIP-V2.0-2021-11-17.pdf`.

[4] Pieter Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 1 ed., April 2013. ISBN: 978-1-449-33406-2.

[5] Timo Glastra, "DIDComm over Bluetooth." https://github.com/decentralized-identity/didcomm-bluetooth/blob/main/spec.md, Feb 2021.

[6] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, P. R. Young, and P. J. Denning, "Computing as a Discipline," *Commun. ACM*, vol. 32, p. 9–23, January 1989.

[7] M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, and C. Allen, "Decentralized Identifiers (DIDs) v1.0." `https://www.w3.org/TR/did-core/`, July 2022.

[8] M. Sporny, D. Longley, and D. Chadwick, "Verifiable Credentials Data Model v1.1." `https://www.w3.org/TR/vc-data-model/`, March 2022.

[9] R. C. Joseph, "Data Breaches: Public Sector Perspectives," *IT Professional*, vol. 20, no. 4, pp. 57–64, 2018.

[10] C. Timberg, "A flaw in the design," *The Washington Post*, vol. 1, May 2015. `https://www.washingtonpost.com/sf/business/2015/05/30/net-of-insecurity-part-1/`.

[11] N. Meulen, "DigiNotar: Dissecting the First Dutch Digital Disaster," *Journal of Strategic Security*, vol. 6, pp. 46–58, June 2013.

[12] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The cost of the "s" in https," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, (New York, NY, USA), p. 133–140, Association for Computing Machinery, 2014.

[13] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, RFC Editor, August 2018.

[14] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, pp. 1–7, Athens, Greece, 2011.

[15] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano, "PGM Reliable Transport Protocol Specification," RFC 3208, RFC Editor, December 2001. `http://www.rfc-editor.org/rfc/rfc3208.txt`.

[16] K. D. Hartog, Stephen Curran, Sam Curren, and M. Lodder, "Aries RFC 0019: Encryption Envelope." `https://github.com/hyperledger/aries-rfcs/tree/main/features/0019-encryption-envelope`, May 2019.

[17] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," RFC 7748, RFC Editor, January 2016.

[18] D. J. Bernstein, T. Lange, and P. Schwabe, "The Security Impact of a New Cryptographic Library," in *Progress in Cryptology – LATINCRYPT 2012* (A. Hevia and G. Neven, eds.), (Berlin, Heidelberg), pp. 159–176, Springer Berlin Heidelberg, 2012.

[19] R. West, D. Bluhm, M. Hailstone, Stephen Curran, Sam Curren, and G. Aristy, "Aries RFC 0023: DID Exchange Protocol." `https://github.com/hyperledger/aries-rfcs/tree/main/features/0023-did-exchange`, April 2021.

[20] E. Wilde and A. Vaha-Sipila, "URI Scheme for Global System for Mobile Communications (GSM) Short Message Service (SMS)," RFC 5724, RFC Editor, January 2010.

[21] R. West, D. Bluhm, M. Hailstone, Stephen Curran, Sam Curren, and G. Aristy, "Aries RFC 0434: Out-of-Band Protocol 1.1." https://github.com/hyperledger/aries-rfcs/tree/main/features/0434-outofband, March 2020.

[22] Oliver Terbu, "need repeatable methodology for new transports," Github, Decentralized Identity Foundation, Aug 2021. `https://github.com/decentralized-identity/didcomm-messaging/issues/222#issuecomment-904719436`.

[23] S. Curren, T. Looker, and O. Terbu, "DIDComm Messaging v2.0," tech. rep., Decentralized Identity Foundation. `https://identity.foundation/didcomm-messaging/spec/v2.0/`.

[24] Daniel Hardman, "Aries RFC 0031: Discover Features Protocol 1.0." https://github.com/hyperledger/aries-rfcs/tree/main/features/0031-discover-features, May 2019.

[25] A. Cooper, H. Tschofenig, B. Aboba, J. Peterson, J. Morris, M. Hansen, and R. Smith, "Privacy considerations for internet protocols," RFC 6973, RFC Editor, July 2013.

[26] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001* (R. Guerraoui, ed.), (Berlin, Heidelberg), pp. 329–350, Springer Berlin Heidelberg, 2001.

[27] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *Networked Group Communication* (J. Crowcroft and M. Hofmann, eds.), (Berlin, Heidelberg), pp. 30–43, Springer Berlin Heidelberg, 2001.

[28] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, vol. 19, p. 332–383, August 2001.

[29] R. Van Renesse, K. P. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 76–83, 1996.

[30] Pieter Hintjens, "37/ZMTP ZeroMQ Message Transport Protocol," ZeroMQ RFC 37/ZMTP, iMatix Corporation. https://rfc.zeromq.org/spec/37/.

[31] R. Moats, "Urn syntax," RFC 2141, RFC Editor, May 1997.

[32] Gregory Szorc, "Better Compression with Zstandard." https://gregoryszorc.com/blog/2017/03/07/better-compression-with-zstandard/, March 2017.

[33] P. J. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," RFC 4122, RFC Editor, July 2005. http://www.rfc-editor.org/rfc/rfc4122.txt.

[34] Y. Polyakov, K. Rohloff, G. Sahu, and V. Vaikuntanathan, "Fast proxy re-encryption for publish/subscribe systems," *ACM Transactions on Privacy and Security*, vol. 20, September 2017.

[35] Daniel Grosu, "Some Performance Metrics for Heterogeneous Distributed Systems," vol. III, pp. 1261–1267, Aug 1996.

[36] M. Caporuscio, A. Carzaniga, and A. Wolf, "Design and evaluation of a support service for mobile, wireless publish/subscribe applications," *IEEE Transactions on Software Engineering*, vol. 29, no. 12, pp. 1059–1071, 2003.

[37] Francesco, "[zeromq-dev] Inefficient TCP connection for my PUB-SUB zmq communication," Internet Forum, The Mail Archive, March 2021. https://www.mail-archive.com/zeromq-dev@lists.zeromq.org/msg31273.html.

[38] Luca Boccassi, "[zeromq-dev] Message batching in zmq," Internet Forum, The Mail Archive, Aug 2019. https://www.mail-archive.com/zeromq-dev@lists.zeromq.org/msg30794.html.

[39] Amy Brown and Greg Wilson, "The architecture of open source applications," vol. II, ch. ZeroMQ, Creative Commons, 2012. ISBN: 978-1257638017.

[40] J. Oikarinen and D. Reed, "Internet relay chat protocol," RFC 1459, RFC Editor, May 1993. http://www.rfc-editor.org/rfc/rfc1459.txt.

[41] "zmq_setsockopt(3)," ZeroMQ API ØMQ/4.0.9, iMatix Corporation. http://api.zeromq.org/4-0:zmq-setsockopt.

[42] Pieter Hintjens, "ZeroMQ Authentication Protocol," RFC 27/ZAP, ZeroMQ RFC. https://rfc.zeromq.org/spec/27/.

[43] Pieter Hintjens, "Using ZeroMQ Security (part 1)." http://hintjens.com/blog:48, September 2013.

[44] F. Lau, S. Rubin, M. Smith, and L. Trajkovic, "Distributed denial of service attacks," in *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0*, vol. 3, pp. 2275–2280 vol.3, 2000.

[45] Mike Hanley, "We updated our RSA SSH host key." https://github.blog/2023-03-23-we-updated-our-rsa-ssh-host-key/, March 2023.

[46] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, IEEE, 1994.

[47] D. Micciancio and O. Regev, *Lattice-based Cryptography*, pp. 147–191. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[48] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001.

[49] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems," *SIGPLAN Not.*, vol. 51, p. 371–384, January 2016.

[50] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, p. 179–196. New York, NY, USA: Association for Computing Machinery, 2019.

[51] "zmq_msg_init_size(3)," ZeroMQ API ØMQ/4.3.2, iMatix Corporation. `http://api.zeromq.org/master:zmq-msg-init-size`.

[52] M. Sustrik, "Internal Architecture of libzmq," white paper, November 2010. `http://wiki.zeromq.org/whitepapers:architecture`.

[53] Pieter Hintjens, "ZeroMQ Publish-Subscribe," RFC 29/PUBSUB, ZeroMQ RFC. `https://rfc.zeromq.org/spec/29/`.

[54] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing," *Communications of the ACM*, vol. 42, no. 2, pp. 39–41, 1999.

[55] Daniel Hardman, "Aries RFC 0046: Mediators and Relays." `https://github.com/hyperledger/aries-rfcs/blob/main/concepts/0046-mediators-and-relays/README.md#summary`, February 2019.

[56] P. Rösler, C. Mainka, and J. Schwenk, "More is less: on the end-to-end security of group chats in signal, whatsapp, and threema," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 415–429, IEEE, 2018.

[57] Daniel Hardman, Sam Curren, Andrew Whitehead, "Aries RFC 0017: Attachments." `https://github.com/hyperledger/aries-rfcs/blob/main/concepts/0017-attachments/README.md`, January 2019.