NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

GRADUATE PROGRAM
“COMPUTING SYSTEMS: SOFTWARE AND HARDWARE”

MASTER OF SCIENCE THESIS

A Generic Connectivity Service for peer-to-peer applications

Christos P. Aslanoglou

Supervisors: **Mema Roussopoulos,** Associate Professor
Nikos Chondros, Postdoctorate researcher
Michalis Konstantopoulos, Ph.D. candidate

ATHENS

DECEMBER 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
“ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γενικευμένο Σύστημα Διασύνδεσης για εφαρμογές ομοτίμων χρηστών

Χρήστος Π. Ασλάνογλου

Επιβλέποντες: **Μέμα Ρουσσοπούλου,** Αναπληρώτρια Καθηγήτρια
Νίκος Χονδρός, Μεταδιδακτορικός ερευνητής
Μιχάλης Κωνσταντόπουλος, Υποψήφιος διδάκτωρ

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΗΣ 2019

MASTER OF SCIENCE THESIS

A Generic Connectivity Service for peer-to-peer applications

Christos P. Aslanoglou
S.N.: M1468

SUPERVISORS: **Mema Roussopoulos,** Associate Professor
Nikos Chondros, Postdoctorate researcher
Michalis Konstantopoulos, Ph.D. candidate

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γενικευμένο Σύστημα Διασύνδεσης για εφαρμογές ομοτίμων χρηστών

Χρήστος Π. Ασλάνογλου
Α.Μ.: Μ1468

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Μέμα Ρουσσοπούλου,** Αναπληρώτρια Καθηγήτρια
Νίκος Χονδρός, Μεταδιδακτορικός ερευνητής
Μιχάλης Κωνσταντόπουλος, Υποψήφιος διδάκτωρ

ABSTRACT

With the advent of cloud computing, all data flows among users of applications follows a de facto route, through third party services. In fact, data, before reaching their recipient, are stored in cloud storage, even though not always needed. Additionally, services on the internet always try to lock-in their users and their friends. In this thesis we present the design, implementation and evaluation of Generic Connectivity Service (GCS), a middleware service facilitating peer-to-peer connection establishment among applications running on users' devices. Simultaneously, GCS aims to give back to users the management of their contacts list. Thus, decoupling applications from handling each users' friends-list.

Our system comprises a network of GCS nodes and a daemon running on each user's device. Users connect to GCS network, thus rendering their devices available for connection requests. Applications running on a user's device request a connection towards one of her friends through the daemon component. In sequence, the daemon forwards a connection request to GCS and if callee accepts the connection, more control messages are exchanged, necessary for connection establishment. Once a peer-to-peer channel is established, the two applications can communicate without their data flowing through GCS nor a third party cloud service.

We have provided both a centralized and a distributed version of our system. The latter of course, mitigates a single point of failure, thus improving our system's availability and fault tolerance. In the distributed version of our system, GCS nodes form a Distributed Hash Table which is used as a routing layer for control messages. Finally, an enticing property of our system is that it allows connection establishment among users even behined NATs and firewalls.

SUBJECT AREA: Distributed Systems

KEYWORDS: Middleware, Distributed Hash Table, Peer-to-Peer Communication

ΠΕΡΙΛΗΨΗ

Με τον ερχομό των νεφών υπολογιστικών συστημάτων, όλες οι ροές δεδομένων ανάμεσα σε χρήστες διαφόρων εφαρμογών ακολουθούν μια διαδρομή διαμέσου τρίτων υπηρεσιών. Τα δεδομένα, πριν φθάσουν στον παραλήπτη τους, αποθηκεύκονται σε αποθηκευτικό χώρο υπολογιστικού νέφους, χωρίς αυτό να είναι πάντα απαραίτητο. Σε αυτήν την διπλωματική εργασία παρουσιάζουμε τον σχεδιασμό, την υλοποίηση και την αξιολόγηση του Συστήματος Γενικής Διασύνδεσης (ΣΓΔ), μια κατανεμημένη πλατφόρμα που συντελεί στην δημιουργία καναλιών επικοινωνίας ομοτίμων μεταξύ εφαρμογών που τρέχουν σε συσκευές των χρηστών. Ταυτόχρονα, το ΣΓΔ στοχεύει να επιστρέψει στους χρήστες τον έλεγχο και τη διαχείριση της λίστας των επαφών τους. Με αυτόν τον τρόπο απεμπλέκει τις εφαρμογές από τη διαχείριση της λίστας φίλων του εκάστοτε χρήστη.

Το σύστημά μας αποτελείται από ένα δίκτυο από κόμβους του ΣΓΔ καθώς και μια εφαρμογή τύπου δαίμονα (που τρέχει στο παρασκήνιο), η οποία τρέχει σε κάθε συσκευή των χρηστών. Οι εφαρμογές που τρέχουν στην συσκευή ενός χρήστη, αιτούνται τη δημιουργία καναλιού επικοινωνίας με έναν από τους φίλους του χρήστη διαμέσου της εφαρμογής δαίμονα. Στη συνέχεια, ο δαίμονας προωθεί ένα αίτημα σύνδεσης στο ΣΓΔ κι αν ο παραλήπτης το δεχθεί, τότε θα ανταλαχθούν κάποια ακόμη μηνύματα ελέγχου, απαραίτητα για την δημιουργία του καναλιού. Μόλις το κανάλι επικοινωνίας έχει εγκαθιδρυθεί, οι δύο εφαρμογές επικοινωνούν χωρίς τα δεδομένα τους να διέρχονται διαμέσου του ΣΓΔ καθώς και τρίτων υπηρεσιών υπολογιστικού νέφους.

Παρέχουμε δύο εκδόσεις του συστήματός μας, την κεντρικοποιημένη και την κατανεμημένη. Η κατανεμημένη έκδοση, αποφεύγει το πρόβλημα του μοναδικού σημείου αποτυχίας, παρέχοντας έτσι, βελτιωμένη διαθεσιμότητα του συστήματος αλλά και ανοχή σε σφάλματα. Στην κατανεμημένη έκδοση του συστήματος μας, οι κόμβοι του ΣΓΔ σχηματίζουν έναν Κατανεμημένο Πίνακα Κατακερματισμού, ο οποίος χρησιμοποιείται ως επίπεδο δρομολόγησης μηνυμάτων ελέγχου. Τέλος, ένα ακόμη θετικό χαρακτηριστικό του συστήματός μας είναι πως επιτρέπει δημιουργία καναλιών ομοτίμων μεταξύ χρηστών που βρίσκονται σε δίκτυο με μεταφραστή διεύθυνσης δικτίου αλλά και τοίχος προστασίας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κατανεμημένα Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ενδιάμεσο λογισμικό, Κατανεμημένος Πίνακας Κατακερματισμού, Επικοινωνία Ομοτίμων

ACKNOWLEDGEMENTS

Για τη διεκπεραίωση της παρούσας Διπλωματικής Εργασίας, θα ήθελα να ευχαριστήσω τους επιβλέποντες καθηγήτρια Μέμα Ρουσσοπούλου, μεταδιδακτορικό ερευνητή Νίκο Χονδρό και υποψήφιο διδάκτορα Μιχάλη Κωνσταντόπουλο, για την συνεργασία, την καθοδήγησή τους και την πολύτιμη συμβολή τους στην περάτωσή της.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The extended use of cloud services, like Facebook Messenger, Dropbox, Google Drive, has shifted many functionalities towards the cloud, even when it is not mandatory for them to reside there. For example, when friends need to share some files, these have to be first uploaded to a third party service and then shared back to people in the same room. In this case, files, be it sensitive documents, have to go through the cloud, stored there and then downloaded to another device. This model of communication has lead to a reinforcement of the client-server model, which is not always the most appropriate choice. With the advent of high bandwidth connectivity for both, mobile and not, devices, the model can now shift to a hybrid one, whereby client-server and peer-to-peer communication models may be used in tandem, wherever one is more appropriate.

In a world of ubiquitous communication, people choose to contact a friend, a colleague, i.e., another person and not a specific device of a friend. Thus, it seems appropriate to follow the same model when using networked applications. For example, Alice wants to share some files with Bob and does not care on which device her friend receives them. It is generally accepted/known that technology adoption is wider when it is accessible/easy for many people to use it. For this reason, we deem important to follow a more user-centric approach in networked applications, by providing abstractions familiar to users, like that of another user, and not *the current IP address of Bob's laptop*.

On the subject of easily usable technology, another problem of current networked applications is the "lock-in factor". To elaborate, most companies want to lock-in more customers to their products, thus provide full fledged solutions, such as online file storage, along with collaborative text-editing and email. Thus, a group of colleagues who need to collaborate on a project, all have to have an account on such a service but also accept other group members as contacts. The problem becomes more obvious if they require an additional service, for example photo editing. Each group member has to have an account at that service and also accept others as contacts or "friends". One can notice how ineffective this becomes as the number of required services increases.

In this work, we present a distributed connectivity service, which facilitates peer-to-peer connection establishment among applications running on users' devices. Our system, *Generic Connectivity Service (GCS)* aims to provide networked applications an easy way to communicate with other applications, by avoiding cloud services when not necessary. In GCS, a user is a first-class citizen. A user's identity is defined by using a Blockstack Naming System (BNS) [12] identity or a public key. The user has devices, with which she connects to GCS, thus making them available for incoming connection requests.

A user defines friends, i.e., a white-list of contacts that are allowed to request incoming connections. Through this way, we decouple friends management from applications and move it from companies ownership to the user's discretion, that is, the user is the owner of her own friends-list. In the aforementioned scenario of a group of colleagues, the set of required accounts stays constant regardless of the number of required collaboration services. The system also allows the user to select among three privacy modes regarding her friends-list. A user may choose not to disclose who her contacts are, or she may upload her friends-list to GCS. We expand on the privacy modes on section 4.2.

GCS provides an abstraction for P2P connections and the required complexity of establishing one regardless of the peers' network topologies. GCS, apart from a distributed service, is accompanied by a daemon that runs on a user's device and mediates connec-

tion requests from/to local applications. Applications are only required to use the daemon's exposed API for requesting available devices of a user, her friends and finally requesting the establishment of a P2P connection. We describe the exposed API on section **??**. The system can establish P2P connection even when a peer is behind a firewall or a NAT.

The contributions of this work are as follows:

- GCS is a distributed platform for facilitating peer-to-peer connection establishment.
- Introduces the user as a first-class citizen of networked applications. Thus, users can easily select other friends to collaborate with, instead of having to deal with IP addresses.
- Decouples the management of contacts lists from applications, thus making the user the owner of her friends-list.
- A distributed middleware system without a single point of failure.

2. RELATED WORK

Several connectivity establishment systems have been proposed, although most of them are not designed around the notion of communication among users, through their devices. Related work includes systems which use centralized components to resolve domain names using DNS infrastructure (NUTSS [19], Signpost [30]). Other systems have centralized components and create a rendezvous service for either propagating messages or establishing peer-to-peer connections (N2N [15], HeNNA [26], TinCan [32]) or require an always-on and reachable device (Signpost [30]). Finally, there are systems which try to solve the problem in a decentralized manner (UIA [16], DevCom [20], SOS [13]).

In GCS, the user is a first-class citizen, who takes full ownership of her social contacts and is not required to have an always-on device. The system facilitates connectivity establishment and through its decentralized design avoids having a single point of failure. None of the aforementioned systems provide the following features as a whole:

- Allow both communication of personal device and communication among different individuals simultaneously
- Decouple friends management from applications, i.e., making the user the “keeper” of her social contacts
- Have no single point of failure

2.1 Systems with centralized components

NUTSS [19] follows a name based approach for identifying remote hosts and establishing network flows through middleboxes, like NATs and firewalls. NUTSS architecture contains three main components, Policy-Aware Boxes (P-Boxes), Middle-Boxes (M-Boxes) and at its core employs DNS. Each network has a logical P-Box, which in turn is connected to a parent P-Box (that of another network). All P-Boxes form a tree with DNS being its root. When an end-host wants to bind a name to an address it contacts the local P-Box, which in turn forwards the request to its parent, up until the *contact* P-Box (connected to DNS). The latter, uses DNS queries to identify the location of a device.

The next step is establishing a connection and involves M-Boxes. M-Boxes are located on the network boundary and are associated with a P-Box. A network path is established through M-Boxes, to which end-hosts send encrypted data. GCS does not use DNS, it does not need network wide changes (like the presence of P-Boxes and M-Boxes) and it facilitates P2P channel establishment, not a network path from one device to another.

N2N [15] is a peer-to-peer overlay, formed by edge nodes and super nodes. The latter are used to introduce edge nodes and for crossing symmetric NAT (thus should be publicly accessible). Users can form and take part in multiple communities whose membership is static. Each community is a different network interface and is identified through its MAC address.

In contrast to GCS, N2N not only lacks a user concept, but also imposes a mental burden of having to remember which network interface to use for a given application and friend. In our system, users solely need to specify a friend’s name and possibly choose one of their friend’s available devices.

HeNNA [26] decouples identification from location and allows message delivery across heterogeneous networks, including infrastructure-based and ad-hoc networks, while coping with nodes of intermittent connectivity. A source does not care about the current location (IP address) of a destination node. The latter may be connected to any network using any interface at the time of message arrival. Thus, applications bind to node identifiers instead of IP addresses. Additionally, each node's location information is managed by an always reachable node, namely Location and Management Server (LMS). LMS has a globally reachable address and maintains location information of registered nodes. The authors suggest LMS can be hosted by an ISP, a company or a user for her personal devices. Finally, this node may store messages on behalf of unaivalable nodes. Compared to GCS, HeNNA assumes an always reachable node for each (privacy conscious) user. There's also no peer-to-peer secure channel and social contacts list for white-listing connection requests.

Signpost [30] is a network architecture which like GCS provides a rendezvous service. The system's goal is to map a domain name hierarchy (DNS zone) to a user's personal cloud of devices. Each user has a Signpost Controller, which stores a zone file with domain name and device associations. The controller is an always-on and publicly accessible node, as it responds to DNS queries for devices registered in the zone file. Compared to GCS, a user is required to acquire a DNS domain and own a 24/7 publicly available device, which is also a single point of failure.

TinCan [32] facilitates the creation of peer-to-peer virtual private networks. Its goals differ from these of GCS, but the proposed design is similar. TinCan uses XMPP [31] for discovery/notification, STUN [34] for reflection and TURN [23] for relaying. The system offers two possible VPN topologies, *Group VPN* and *Social VPN*.

In the case of *Group VPN*, users/devices have to share credentials for the XMPP server and the network creator has to assign each device a virtual address. A usecase of *Group VPN* is for communication among personal devices of a user or users with no identities in social networks. In the *Social VPN* case, trust relationships are maintained by centralized (or, federated) servers (e.g. Google Hangouts [6]) and only users that are trusted under these services can establish P2P connections. For example, if Alice has a Facebook identity, while Bob has a Google identity, they will not be able to reach each other (except for a setup for XMPP federation). If Alice decides to delete her Facebook identity, she will not have access to her buddy list, as it is maintained by the server.

Compared to GCS, TinCan provides independent/isolated solutions for two problems. Alice can utilize *Group VPN* for communication among her personal devices, while she can communicate with Bob using *Social VPN*. However, given Alice and Bob owning more than one devices, two problems arise. It is not possible for Alice's laptop to choose which of Bob's devices to contact, while also not feasible for Bob to instruct the system to which device he wants to be reached at certain times.

2.2 Distributed Systems

UIA [16] is a distributed system that shares goals with GCS. Each user has a namespace and each device has a unique endpoint identifier (EID) in the system. Namespaces are shared across all users and each network change is gossiped. Devices run a handshake protocol to authenticate each other and form/merge groups. When a device has to connect to another device, namely callee, it aims to find its address. Initially, it tries to contact

callee's last known addresses. If that fails, the caller queries other known devices using callee's EID.

In contrast to GCS, UIA reveals information about each user's devices under her namespace, her social contacts (as part of other namespaces), it has not implemented NAT traversal for establishing P2P connections and it assumes a high number of UIA users so that routing among two devices is always possible.

DevCom [20] is another distributed network system, which assists users to easily organize devices in multiple trustworthy communities. The authors envision users participating in multiple, concurrent communities. The latter are comprised of devices and have common trust policies where communication, sharing and collaboration among devices are assured. DevCom uses static virtual IP addresses, so that even on connectivity changes, the applications can use the same address.

A device can detect another using either mDNS, DNS Service Discovery, NFC, QR codes or remotely using email. During discovery public keys are exchanged so that devices can then establish secure communication channels. Devices maintain a list of all other devices with which they share a community and also propagate community membership changes to others. Two devices may be known with more than one DevCom IPs, if they are participating in more than one common communities.

To connect to a device, the physical address is needed. A list of history addresses is maintained for each trusted device, and if none of these are valid, it tries to identify the current address through other community members. The authors claim DevCom can be used with legacy applications, as it uses IPs and there is zero configuration required by users. However, users have to identify and retrieve the IP of the device with which they wish to communicate.

Compared to GCS, like TinCan, communication is among devices and not user identities. Additionally, DevCom assumes that at any time at least one device in the community can be reached in a previously used physical IP. Finally, the system requires direct control channel between every online device of a community.

In **Secure Opportunistic Schemes (SOS)** [13] the authors outline a communications middleware for Apple devices. When an application (which uses SOS) is run for the first time, a public/private key pair is created and a valid certificate is signed by the Certificate Authority of the application. Each application is then able to announce messages and any devices interested in these messages can request the content. The requesting device also sents the signed certificate, which contains its public key, to verify trust and establish a secure channel. A device may store messages of other users and announce them along with its own, thus acting as a proxy. SOS in contrast to GCS, is only applicable to Apple devices.

3. PRELIMINARIES

3.1 Interactive Connectivity Establishment

Our system's aim is to facilitate peer-to-peer channel establishment among applications running on various network topologies. This goal entails being able to traverse across different NAT topologies, even behind firewalls.
The standard for such requirements is a family of protocols, namely Interactive Connectivity Establishment (ICE) [29]. ICE works by exchanging all possible IP addresses and ports which are then tested for connectivity by peer-to-peer connectivity checks. These IP addresses and ports are then exchanged through GCS from a caller to a callee, and vice versa. Finally, connectivity checks are performed using Session Traversal Utilities for NAT (STUN) [28] and Traversal Using Relays around NAT (TURN) [22] protocols orchestrated by ICE.

In a typical ICE scenario, there are two ICE agents (endpoints), whose aim is to communicate. ICE assumes there's a way to exchange signaling information among the agents. Initially, the agents are unaware of their network topology. ICE allows the agents to discover required information about their topologies and thus find one or more paths through which they establish a data session. In a standard deployment of ICE, apart from the two ICE agents, there's the signaling server and the servers running STUN and TURN protocols.

A high level overview of how ICE works follows. Each agent gathers the variety of candidate transport addresses (IP address and port for a particular protocol) with which it can communicate with the other agent. These candidate addresses may include:

- A transport address on a directly attached network interface
- A translated transport address on the public side of a NAT (a "server reflexive" address)
- A transport address allocated from a TURN server (a "relayed address")

Potentially, any of the transport addresses of the caller can be used to communicate with any of the callee's transport addresses. However, not all combinations will work. For example, if both caller and callee are behind NATs, their directly attached interface addresses will not work. The purpose of ICE is to discover the candidate pairs that work. This is done by testing all of the candidate pairs after these are sorted systematically according to metrics defined in ICE, up until one or more pairs allow a data channel to be established.

3.1.1 Candidates

Candidates can be gathered from all available transport addresses. In addition to these, STUN and TURN protocols are also used to obtain additional candidates. STUN provides transport addresses on the public side of a NAT, namely *server reflexive candidates*. This is achieved by the STUN server sending the agent's public address (transport address after NAT) back to the agent, as is demonstrated in figure 3.1.

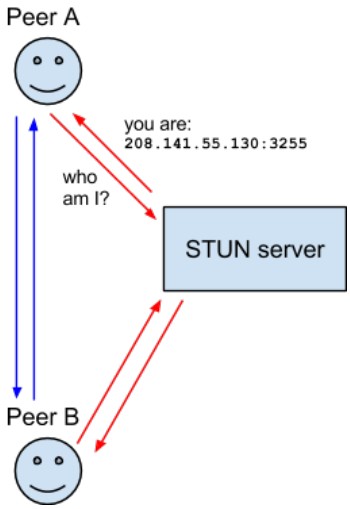

Figure 3.1: Session Traversal Utilities for NAT (STUN) example

There are some network topologies where using STUN does not suffice. An example of this problematic topology is a symmetric NAT. TURN can be used to facilite communication in such cases. TURN provides candidates on TURN servers, namely *relayed candidates*. When TURN is used, an agent allocates an address to the TURN server, which can be used as a candidate. Thus, a TURN server acts as a relay as show in figure 3.2.

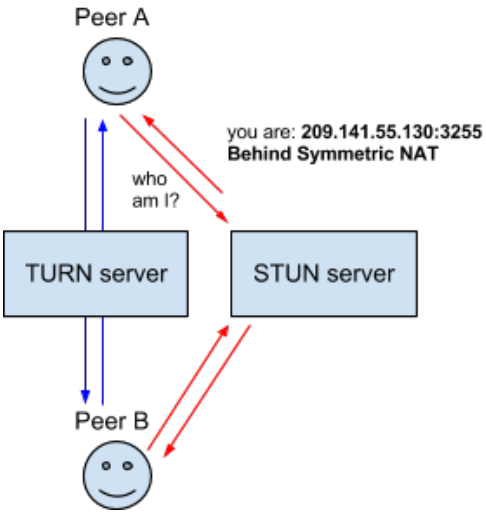

Figure 3.2: Traversal Using Relays around NAT (TURN) example

This process requires both STUN and TURN servers to be publicly accessible to the internet.

ICE will eventually find a working pair of candidates through which communication can commence. To shorten this process, the candidates are sorted. The basis for sorting candidate pairs is that more direct routes (i.e., transport addresses on a directly attached network interface) are prefered over ones involving more hops (such as TURN servers). In addition to the above, same type of candidate addresses are assigned same priorities.

Candidates are exchanged through a signaling service. These can be encoded in Session Description Protocol (SDP, [21]) offers.

3.1.2 Trickle ICE

The aforementioned process for gathering candidates, sorting them and then exchanging them may result in a lengthy connection establishment process and degraded user experience. Thus, an improvement has been suggested, known as “Trickle ICE”. According to this enhancement of the ICE protocol, candidates are exchanged as soon as an ICE session has been initiated. Trickle ICE allows for shorter session establishment times, as allows connectivity checks for candidate pairs to be run in parallel.

3.2 Kademlia Distributed Hash Table

In computer systems, a hash table is a data structure that implements an associative array abstract data type (ADT). This ADT maps keys to values. A hash table uses a hash function to compute an index of an array of buckets or slots, in which the desired value can be found. Ideally, the hash function will assign each key to a unique bucket, but most hash table designs use an imperfect hash function, which generates the same key for different inputs. This introduces collisions, which must be accommodated, by supporting data structures for each array element or bucket.

A distributed hash table provides a key to value mapping, but spread across a peer-to-peer network of nodes. The $\langle key, value \rangle$ pairs are distributed among participating nodes which share hash table maintainance responsibilities. The distribution of $\langle key, value \rangle$ pairs is done in such a way so as to avoid service disruptions upon node arrivals and departures. The distributed and decentralized nature of such systems allows them to scale better than a centralized version of a hash table running in a server.

Kademlia [24] is a Distributed Hash Table (DHT) used by several public networks [8], such as Kad Network [7], BitTorrent [1], Retroshare [10] and IPFS [14]. Kademlia minimizes the amount of messages nodes must send to discover each other, as this information spreads automatically through node lookups. Kademlia uses parallel and asynchronous queries to avoid timeout delays from failed nodes. The key lookup process explores the network in several steps. Each step either provides nodes closer to the requested key or the requested value. The steps stop once no more closer nodes are found.

Keys are derived from an 160-bit namespace of identifiers. Nodes have identifiers derived from aforementioned namespace. $\langle key, value \rangle$ are stored on nodes whose IDs are “close” to the key as per exclusive or (XOR) distance. Kademlia logically spans nodes as leaaves of a binary tree, which each node’s position determined by the shortest unique prefix of its ID. Each node must have a contact (that is, another node) in each of the subtrees (of the whole binary tree) that don’t contain the node, as we can see in figure 3.3. The routing table allows for each step of the lookup process to half the distance towards the key, which provides an efficient $O(logN)$ time complexity.

3.3 Bitcoin Blockchain

Bitcoin is a decentralized peer-to-peer electronic cash system, which is based on a Proof-of-Work (PoW) consensus algorithm which additionally allows for any user to participate in the voting process, by simply using their computing power.

Bitcoin is essentially a distributed ledger, i.e., containing a distributed collection of trans-

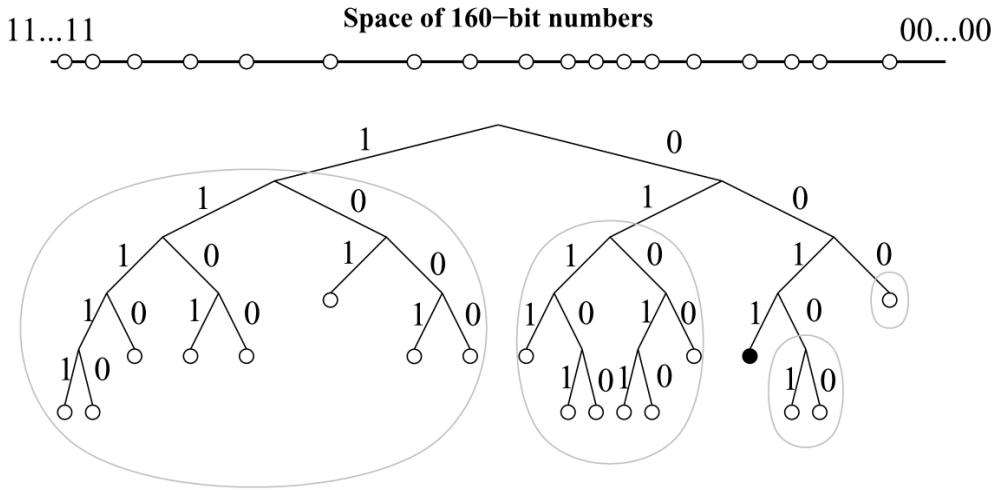

Figure 3.3: Kademlia binary tree. The black dot shows the location of node 0111… in the tree. Gray ovals show subtrees in which node 0111… must have a contact.

actions. The latter ones are contained in blocks, and the latter chained together form the Bitcoin ledger. Chaining occurs by including the hash of the previous block to the next - newly minted - block, acting as an abstraction of a timestamping server.

To render the system as distributed, a consensus algorithm is introduced, namely proof-of-work. Essentially, with transactions broadcast over the network, “miner” nodes combine them to solve a computationally hard puzle. The first node to solve the puzle, broadcasts its solution to the network, so other nodes can verify it. Although, puzzle solving is computationally hard, verification of a solution is fast. Other nodes acknowledge a solution’s validity by simply using it, to “mine” the next block. If there’s a case of forks being created in the ledger (i.e., two nodes solve the puzzle), the protocol’s policy is to use the longest chain or if that’s not enough, the winner block that was created at the earliest time.

To incentivize nodes to support the network, while also, circulating the coins, the first transaction of a block, “mints” new coins owned by the creator (solver) of the block. Transaction fees can be used as an incentive, as well. Fees can happen when a transaction’s output is less that its input value. This can be used to stop minting (at later stages of the currency’s life), as a means of money influx for miners and also mitigating inflation problems.

Finally, although in Bitcoin all transactions are transparent, privacy to the system is introduced by having anonymous wallets. That is, each user can have many wallets which are identified by a public key. Thus, even if transactions are public, money transfers ideally cannot be linked to anyone.

3.4 Blockstack Naming System

Blockstack [12] is a naming system providing human-readable names on top of Bitcoin blockchain [25], as a separate logical layer.

Blockstack authors have separated its data plane from its control plane. The control plane defines the protocol for registering names and creating (name, hash) bindings and consists of the Bitcoin blockchain and a logically separate layer on top, namely *virtualchain*.
The data plane is responsible for data storage and availability. This layer consists of zone files and external storage systems. Zone files allow discovering data by using a hash or a Uniform Resource Locator (URL). External storage systems include any third-party

systems, such as Dropbox [2] and Google Drive [5].

Building systems on top of PoW blockchain cryptosystems, requires tackling limitations on data storage capacity, limited bandwidth and slow writes. The two planes design, increases storage capacity of their system and allows for independent evolution of each layer.

Blockstack is agnostic of the underlying blockchain and provides the ability to construct state machines, through the layer of virtualchains. A virtualchain can introduce new types of state machines without requiring any changes from the underlying blockchain. Transactions (of the blockchain) are treated as input of a virtualchain, whereby valid inputs trigger state changes.

The control plane stores Blockstack operations in the blockchain, such as name registrations and provides a total ordering upon those. The virtualchains layer of the control plane, allows to define new operations, which are encoded in the blockchain layer, as metadata of transactions. Accepted transactions are processed by the virtualchain tier to construct a database that stores info of global system state along with state changes at any blockchain block.

The data plane offers data retrieval routing. Blockstack decouples the task of routing requests (how to find data) from the actual storage of data. This allows using any storage provider and multiple of them to coexist. Zone files, which are identical to DNS zone files (in their format), are used for storing routing information. The virtualchain binds names to hash(zonefile) and stores them in the control plane, while the actual zone files exist on the routing layer. The storage layer of the data plane hosts the actual name-value pairs. Stored data are signed by the key of the respective owner of a name.

In figure 3.4 below we can see the layers of Blockstack's design.

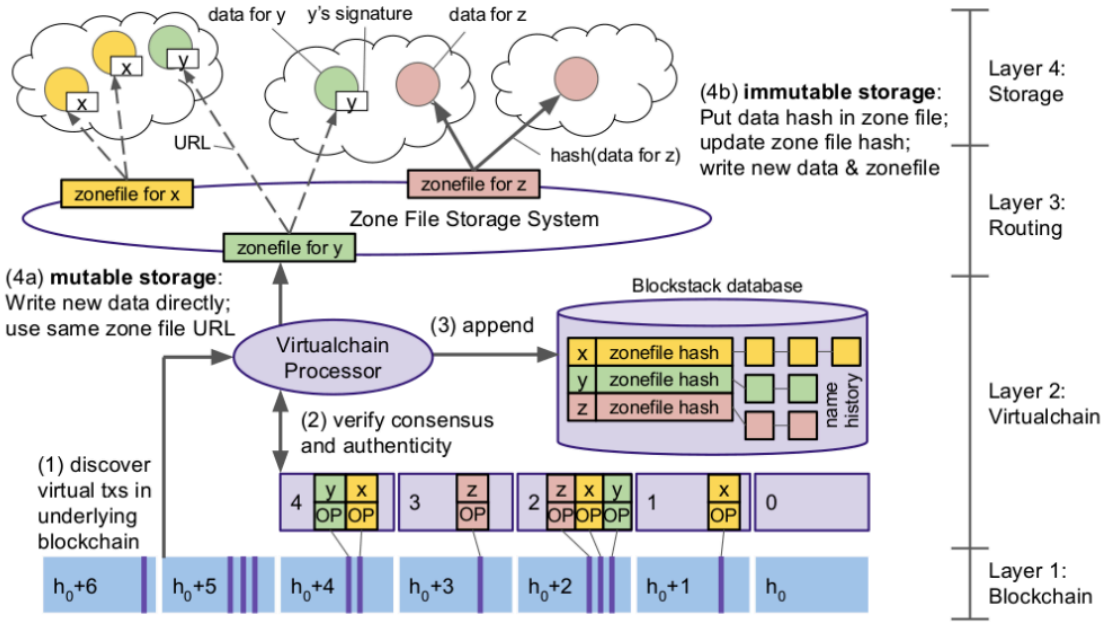

Figure 3.4: Overview of Blockstack's architecture. Blockchain records give (name, hash) mappings. Hashes are looked up in routing layer to discover routes to data. Data, signed by name owner's public-key, is stored in cloud storage.

Concluding, Blockstack provides a complete naming system, whereby names are owned by cryptographic addresses of the underlying blockchain. A user preorders and registers

a name in two steps in order to claim a name without revealing it to the world and thus allowing an attacker to race the user in claiming the name. The first user to successfuly write both a preorder and register transaction is granted ownership of a name.

4. SYSTEM DESIGN

4.1 System Overview

Generic Connectivity Service (GCS) is a distributed platform enabling P2P channel establishment among applications running on end-user devices. Users connect to the service, making available their devices as contact points. In addition, a user defines a friends-list, which acts as a whitelist for users that are allowed to contact her and her devices. The GCS network mediates solely the connection establishment process while the actual communication flows through the P2P channels being handled on each user's device. Below we outline the components of our design. As we can see in figure 4.1, these include:

- A structured network of *GCS* nodes mediating connection establishment among different users and their devices.
- A Contact Management System, namely, *CMS*, acting as the intermediate layer among local applications and the GCS network or another CMS.
- A set of TURN / STUN servers supporting the connection establishment and sometimes the communication, depending on the network topology of each user.

To use the connectivity service, a user needs the CMS running on her devices. One of our system's goals is to allow users take ownership of their friends-list and thus decouple applications from management of such functionality. Thus, not only alleviating the burden of a user having to go through each networked application forming her friends-list, but also gaining some privacy from those applications.

4.2 Generic Connectivity Service

This component handles user authentication and mediates connection establishment. User devices connect to GCS nodes and maintain an open connection to send and receive connection requests. A collection of these nodes form a Distributed Hash Table which acts as a routing layer for control messages. Each GCS node maintains only soft-state which comprises currently connected authenticated clients and their devices. This design decision allows scaling the service by adding more nodes, which simply join the network.
Below we elaborate on the design of a GCS node and what each component achieves, as show in figure 4.2.

Authenticator. Introduces a user to the system through a handshake protocol, which runs when a CMS connects to a GCS node. Users provide their identity to be discoverable by friends. We offer two authentication services:

- public-key cryptography [27], whereby the identity of a user is her public key, and
- Blockstack Naming System (BNS) [12], using the BNS identity of a user. In the latter case, identities are forged on the Bitcoin blockchain [25] and owned by a private key, while also providing a human readable name.

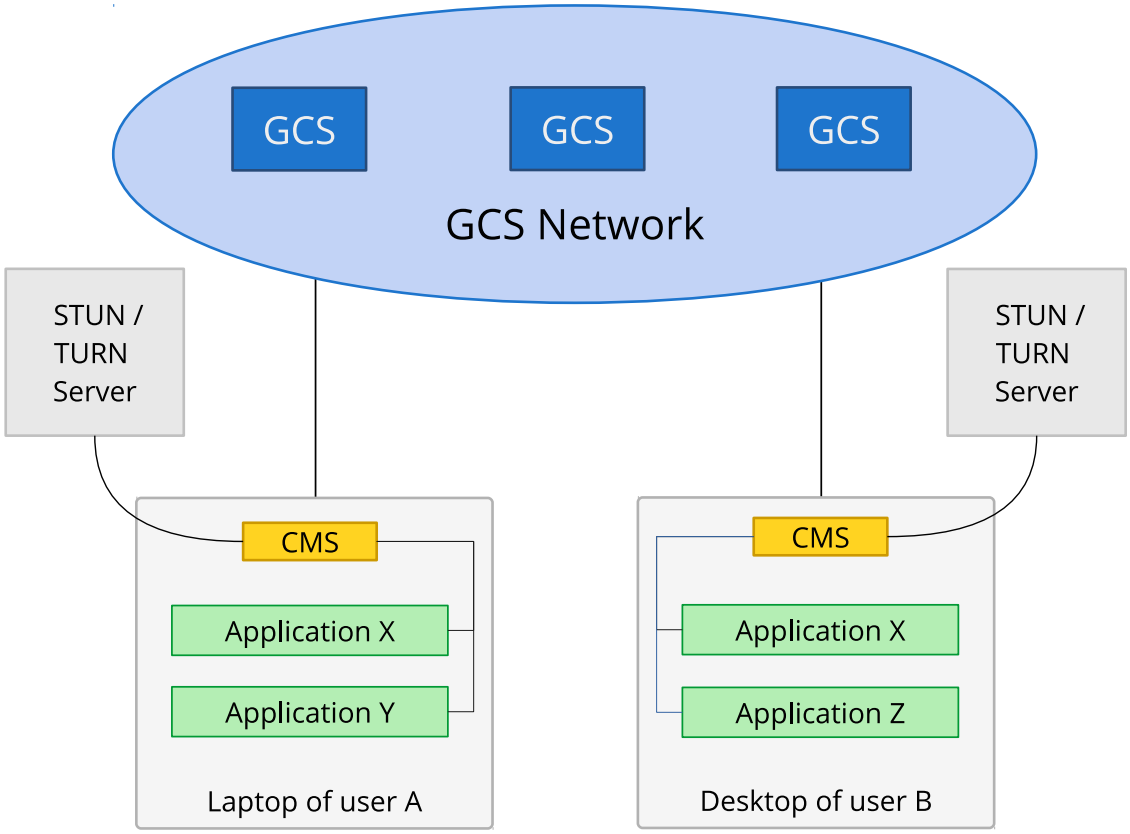

Figure 4.1: Overview of Generic Connectivity Service components.

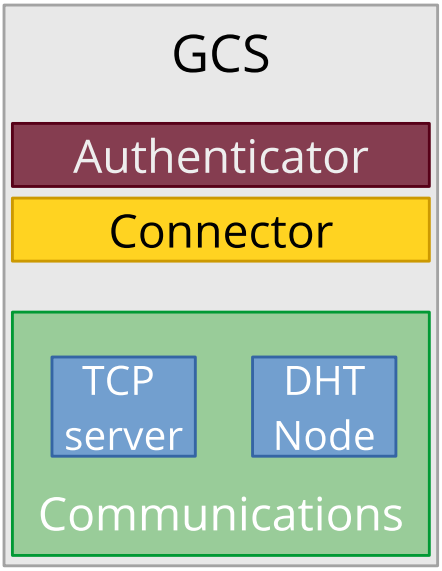

Figure 4.2: Generic Connectivity Service node

During the handshake protocol, users may choose to provide or not their friends-list to a GCS node. As already described, a friends-list acts as a whitelist for users allowed to place connection requests towards the current user. We provide three options for handling a user's friends-list:

- **A user uploads her friends-list.**
  This options provides usability and more security for end-users as GCS nodes cut out spammers, but weakens the privacy model, as a node knows a user's friends.
- **A user doesn't upload her friends-list.**

This achieves some privacy as a GCS node can potentially infer a subset of a user's friends (the ones they accepted a connection request from). Additionally, it incurs bandwidth burdern on a user's device.

- **A user doesn't upload her friends-list & CMSs authenticate over P2P channel.** This option is privacy preserving against an adversary GCS node as the latter cannot infer any of the user's friends. It also provides more guarantees to the end-user as CMSs also authenticate each other over the established P2P channel.

Thus, GCS offers each user an option of "tunable privacy", covering the needs of privacy conscious users up to users who seek usability and minimizing their bandwidth consumption. This trade-off is important to understand as GCS aims to support applications running on either a) devices with no bandwidth limitations (e.g. users' laptop and desktop devices), or b) devices with connectivity subject to provider plans.

Connector. The goal of this component is to mediate connection establishment between a caller and a callee. A CMS requests a connection by providing: 1. identity of the callee, 2. callee's device (identifier), 3. callee's application name and a 4. connection type (data or stream).

To retrieve the available, or currently connected, devices of a callee, we provide the *GetAvailableDevices* protocol (5.3). Regarding the application name parameter, we assume applications running on users' devices share the same identifiers, i.e., a chatting application should use the same identifier regardless of it running on Alice's or Bob's computer. Something like Java's package naming "suggestion" would work, e.g. Google's library Guava identifier is `com.google.guava`, i.e., a reversed internet domain name.

Distributed Hash Table.
To avoid a single point of failure and single point of trust, while also increasing scalability and availability of our system we choose to form a distributed network of GCS nodes. Each node is required to be publicly accessible and is identified using its public key (mapped onto the DHT namespace). Users are also mapped onto the DHT with their identity being the key. Uniqueness of an identity, in the case of BNS identities is guaranteed by the property of the system itself, as it validates upon registration whether a name already exists. While uniqueness of a public key is assumed under standard cryptographic assumptions.

Handler nodes.
A DHT has an identifier namespace which is partitioned among nodes. A user is mapped onto a GCS node's namespace using her identity. CMSs connect to a GCS node from a list well-known (and usually available) nodes. This entails a user may not connect to the node where her identity is mapped. Thus, after a CMS connects to a GCS node and before running the handshake protocol, it requests its $handler$ node in the network. To achieve this, we use the *FindHandlers* protocol (5.1).

4.3 STUN / TURN Server

Our system provides connection establishment even for devices which are not publicly accessible or even behind Network Address Translators (NATs) and firewalls. To achieve this it uses Interactive Connectivity Establishment (ICE) protocol [29] which requires a STUN [34] and a TURN [23] server. In short, a STUN server provides the public IP of a device (behind a NAT), while a TURN server acts as a relay for communication on cases where NAT traversal is infeasible due to the effective NAT topology.

4.4 Contact Management System

This component resides in each user's device and acts as a gateway towards the GCS network. It handles connection establishment requests from local applications and requests coming from remote users through the GCS node. Figure 4.3 demonstrates the components of a CMS.

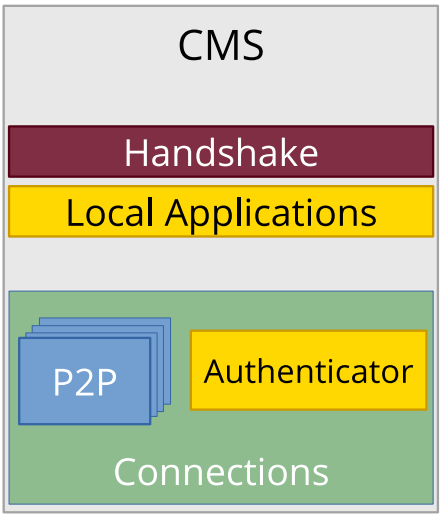

Figure 4.3: Contact Management System

Handshake. Upon startup, CMS connects to a GCS node to find its *handler node*. Once the handler node's address and port are retrieved, the CMS drops the connection with the previous GCS node and initiates a connection with its handler node. Given that, both entities, server and CMS, run the handshake protocol. A successful completion of the protocol signifies that the user is authenticated and has one device available for incoming connection requests. If the same user connects through another device, the same process is followed but now the user has obviously two connected devices ready to accept incoming connection requests. The messages of this protocol are described in 5.2.

Connections. The goal of this component is to setup P2P connections for either requesting local applications or for remote incoming connection request through the GCS node. It achieves this by sending control messages for connection establishment towards its GCS node, which in turn, forwards it to the recipient's GCS node, and finally these reach the callee's CMS. The flow and the messages of this protocol, namely *Connect* are described in 5.3.

This component is also responsible for authenticating both a caller and a callee to each other over a newly established P2P channel. This protects against adversary GCS nodes, who claim a false identity for either the caller and/or the callee. We achieve this through the *Prove Identity* protocol (5.3), which follows a Challenge-Response model. Upon channel establishment, the callee initiates the protocol as a *verifier* and the caller acts as the *prover*. Once this direction of the protocol completes, the roles are swapped and the caller is now the challenger. Upon successful completion of both directions of the protocol, the callee application is notified for an incoming connection request.

Local applications. This component handles all requests from and towards local applications. Applications running on the device connect to the CMS and act as either a server or a client. Server applications communicate their listening port and then listen for clients, as they normally would. Client applications connect to CMS and request a connection

towards a specific user (callee), a (callee) device, an application name and a connection type. Then, our *Connect* protocol (5.3) runs which establishes the requested P2P channel.

To elaborate on the above connection parameters, we assume devices of a user can be distinguished by identifiers assigned by the user, such as *laptop*, or *mobile*. Thus, the tuple $\langle userId, deviceId \rangle$ uniquely identifies a user's device. The connection type parameter describes the channel format and is either 1. data or 2. media stream (audio or video).

Applications communicate with the CMS using an API, which we describe below:

- `GetAvailableDevices(user)`
  Retrieves the currently connected devices of a user (by querying the GCS network).

- `GetFriends()`
  Retrieves the friends (their identities) as defined in the configuration file of the CMS. This decouples friends-lists from applications. Thus a user of a file-sharing application would say *"I want to share this file with Alice.id"*, i.e., using her BNS identity. The application would then request Alice's devices, provide an option to the user and then issue a connection request towards that device.

- `ListenIntent(applicationName, port)`
  Used by server applications to notify their presence to the CMS, by providing their name (or identifier) and port they expect their clients to connect to.

- `Connect(userId, device, application name, connection type)`
  Initiates the *Connect* protocol for establishing a P2P channel. Once the connection has been established, the callee is notified using an `IncomingConnectionRequest(callerUserId, callerDeviceId, callerApplicationName)` message, who may choose to accept or reject the request. Given this response, the caller is notified of the connection request outcome using an `OutgoingConnectionRequestResult` message.

To understand the flow of an application requesting a P2P channel, we provide an example. The caller application may wish to learn the currently connected devices of a callee. Thus, it queries the CMS, which forwards the query to the GCS network. Upon response, the CMS replies to the application with the connected devices of the callee. Caller applications can then issue a connection establishment request to their local CMS. Once the channel has been established, authentication may begin (depending on configuration). Once, authentication succeeds, the callee is notified by its CMS about an incoming connection request. Regardless of whether the callee accepts or rejects the request, the caller is notified of the outcome. If the request was accepted, the CMS (having notified the caller), enters a *message relaying mode*. From that point, the applications can communicate using the P2P channel.

Another property of our system is that it allows communication even between different applications, so for example, a camera application running on a mobile device, could communicate with a file-sharing application of a callee (running on a desktop device), to store a photograph.

5. PROTOCOLS

5.1 Find Handler

When a CMS starts, it has to find the GCS node handling its user identifier. First, it connects to a GCS node from a list of well-known ones. Then, it queries this node for its handler node's transport address, by sending a `GetHandlersForUser` message. The GCS node receiving the query, forwards it through the DHT. This query through the DHT, uses the canonical `FIND_NODES(key)` method, with the key being the user's identifier. Once the reply is received it is forwarded to the CMS via a `DetectedHandlers` message.

In sequence, the CMS drops the connection with the former GCS node and connects to its handler, to which it sends again a `GetHandlersForUser` message. The handler GCS node identifies it should handle the newly connected user, by issuing a query through the DHT. Finally, it replies to the CMS with its transport address. Having completed this phase, both CMS and GCS node are ready to begin the handshake protocol.

5.2 Handshake

The authentication protocol we employ follows a Challenge-Response model, as you can see in figure 5.1. We demonstrate an instance of the protocol using BNS as an authentication service. The protocol's flow using public-key authentication service is the same, except that it does not communicate with Blockstack's infrastructure.

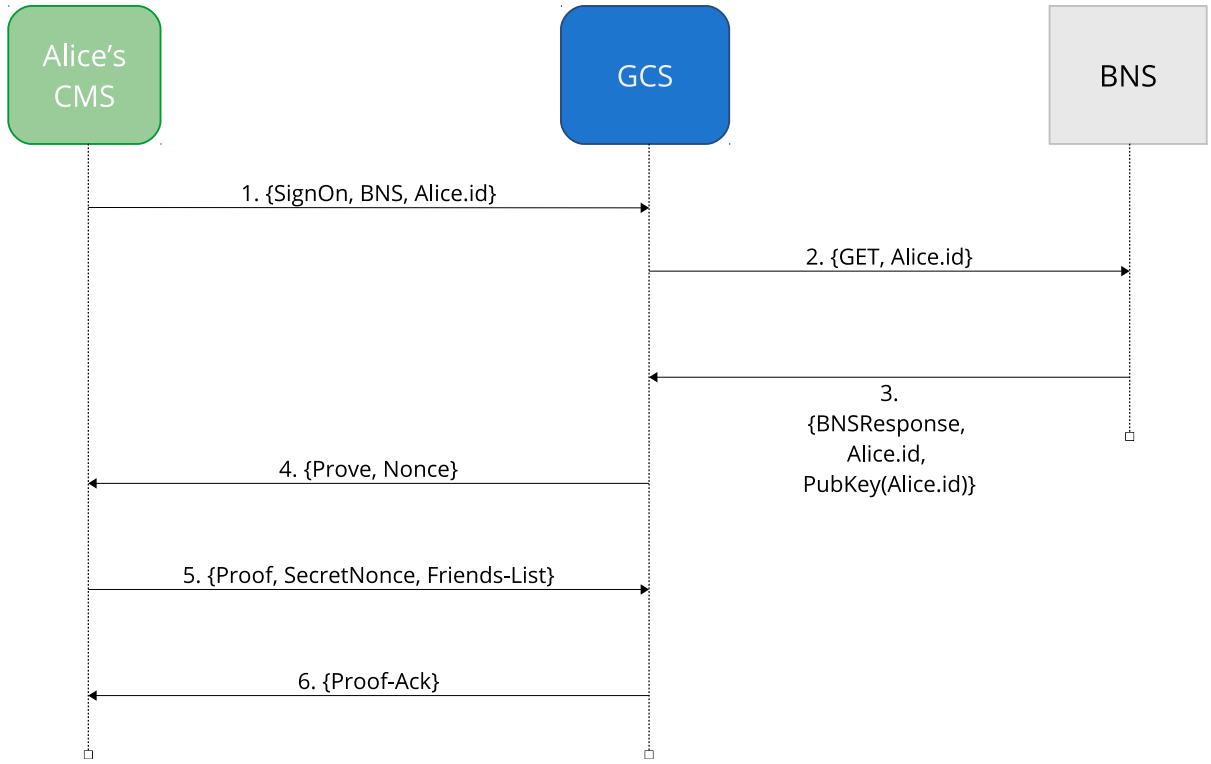

Figure 5.1: Handshake protocol

5.3 Connection

Get Available Devices. Before issuing a connection request, an application may need to query for the available devices of a user. To achieve this, it sends to its local CMS a `GetAvailableDevices(calleeUserId)` message, which in turn forwards it to its GCS node. The GCS node queries the DHT for the handler node of the callee user, and then forwards the request to that node. Having received a reply, it forwards it to the requesting CMS, which then replies to the application. A protocol instance is demonstrated in figure 5.2.

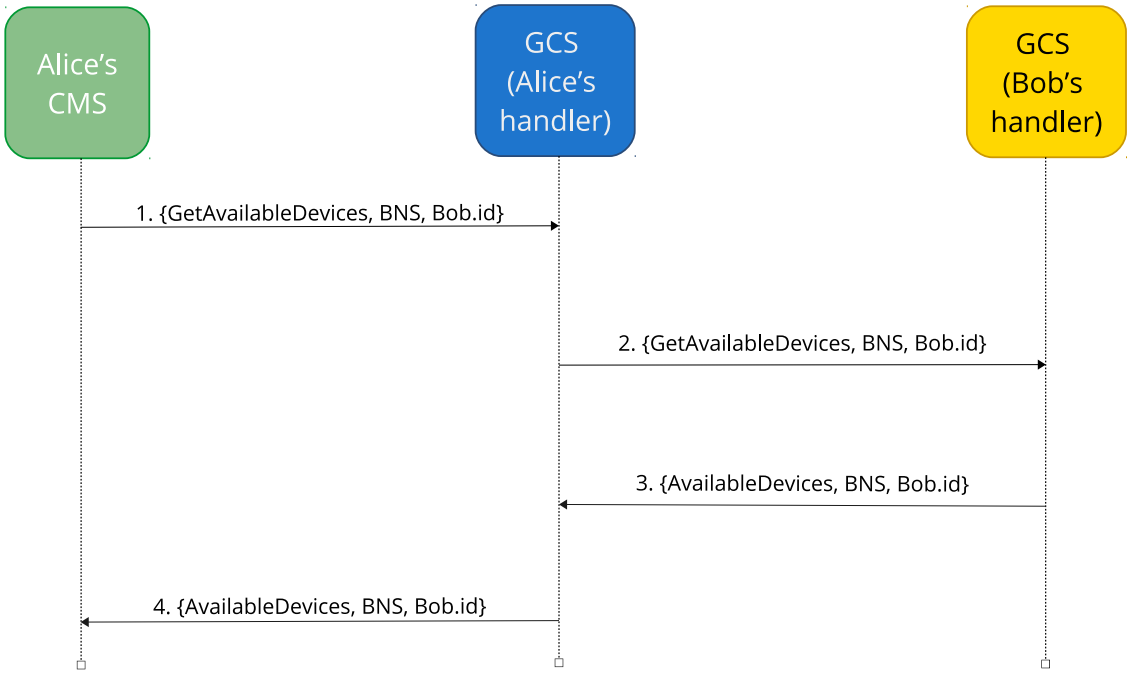

Figure 5.2: Get Available Devices protocol

Connect. The connect protocol is the core of our system, as it establishes P2P channels among requesting CMSs. In figure 5.3, we demonstrate an instance of the protocol spanning two CMSs, that of a caller and a callee and two GCS nodes, the caller's and the callee's handler nodes, respectively. Note that if callee accepts the call, `SignalingData` messages will be sent towards the caller. The same happens towards callee's direction after caller receives `ConnectResult(Accept)`. Multiple `SignalingData` messages may be exchanged among the two parties until a channel is established, depending on their available network interfaces, as ICE protocol [29] instructs.

Prove Identity. A user may choose to authenticate the remote end of a channel, after a P2P connection is established. This is because both GCS and the remote user may be impersonating one of callee's friends. Thus, a user can configure to run the prove identity protocol, whereby both caller and callee are authenticated to each other, over the P2P channel. Note, that this process does not involve GCS at all. In figure 5.4, we show an instance of the protocol, in which Alice has selected to use BNS as an authentication service, while Bob has opted for using his public key.

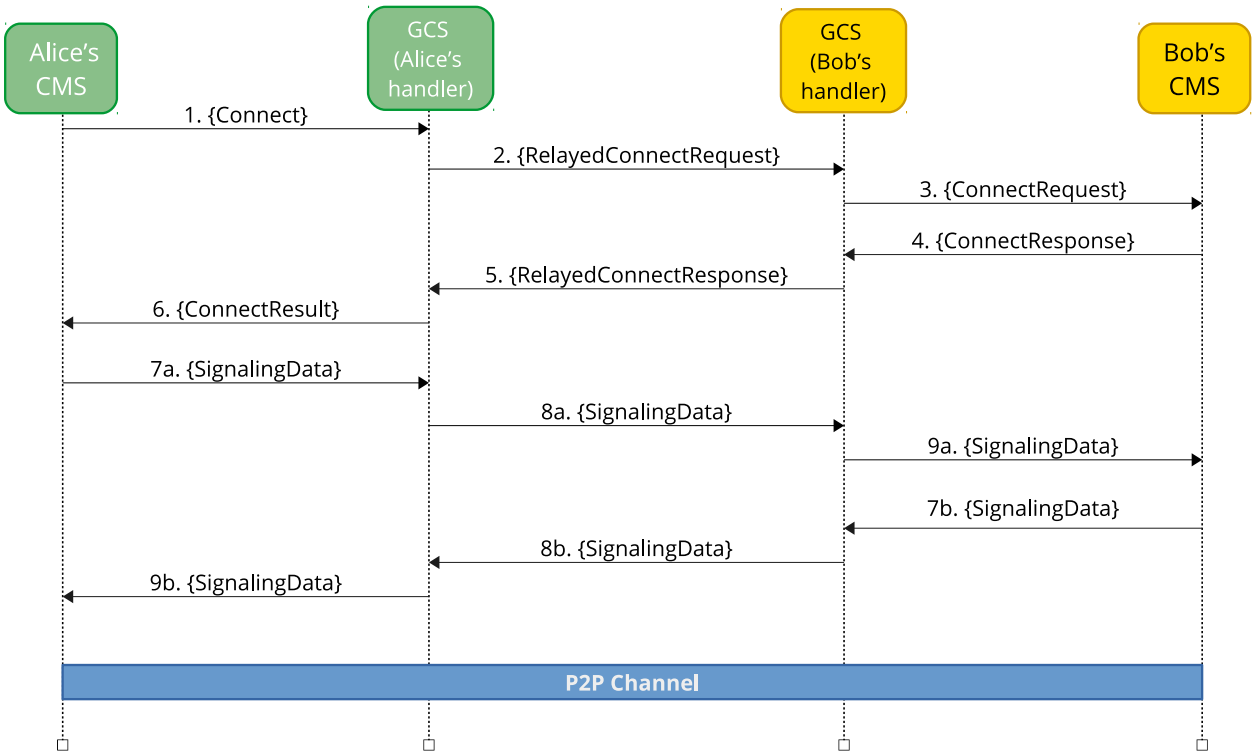

Figure 5.3: Connect protocol

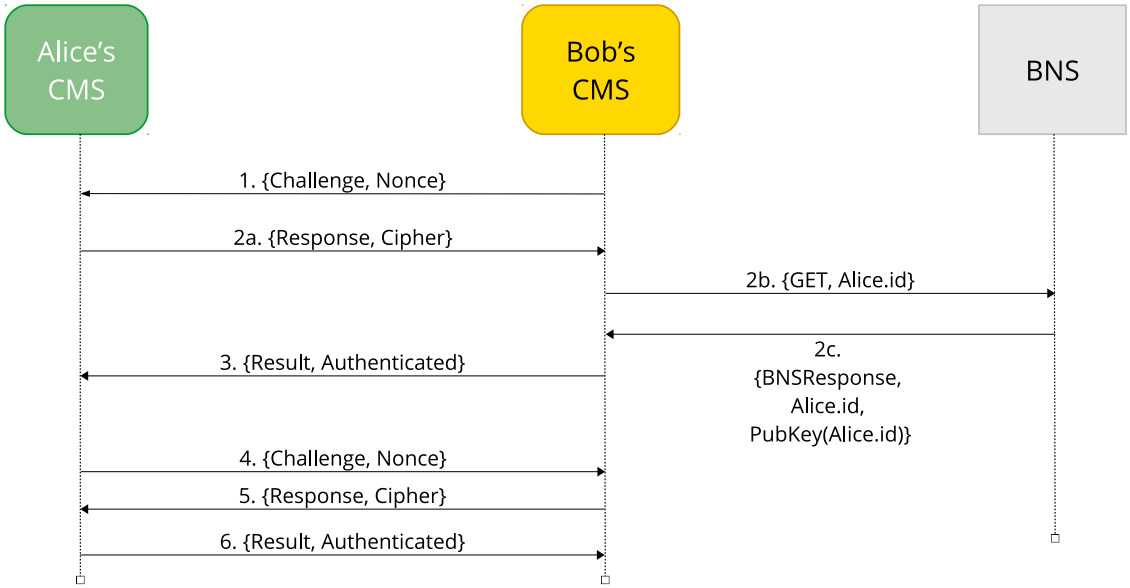

Figure 5.4: Prove Identity protocol

6. EXPERIMENTAL EVALUATION

6.1 Implementation

As illustrated in the system architecture figure (4.1), the system includes a user level CMS and a set of nodes comprising the GCS network. These components are implemented using Javascript running on NodeJS [18].
GCS and CMS communication uses a client-server model using TCP. For message serialization and deserialization we have used FlatBuffers library [3]. FlatBuffers library provides faster (de)serialization and provides access to serialized data without parsing/unpacking, thus is memory efficient, as it only requires the serialized buffer itself to access the data. These capabilities come in the expense of message size, which is a larger against ProtocolBuffers [9] library. For each of our described protocols, we have defined a FlatBuffers schema and then generated using the provided compiler, the serialization and deserialization functions for Javascript.
CMS is both a TCP client for GCS - CMS communication, but also a TCP server for local applications. For communication among CMS and local applications, we chose FlatBuffers. Due to the language agnostic schema of FlatBuffers, we can compile serialization and deserialization helpers for many languages, such as C, C++ and Java.

For peer-to-peer channel creation, we required an implementation of the Interactive Connectivity Establishment (ICE) protocol. To this purpose we used the de facto library for real-time communications, WebRTC [11].

For the distributed version of GCS, which requires a Distributed Hash Table, we have used Kadence [17] which implements Kademlia DHT [24].

To develop these software systems, we followed test driven design methodology, whereby we firstly wrote tests which failed but captured the requirements needed to implement. Then wrote just enough code to make the tests pass and then refactored until we're satisfied with the outcome and design of each module.
Finally, we followed Continuous Integration (CI), which is the practice of integrating code into a shared repository and testing each change automatically, as early as possible. Thus, we have used GitLab [4] as our code repository, which also provides infrastructure to run automated tests in a containerized environment. After pushing a change to the repository, each commit was checked out by a container, our package was installed (installing its dependencies) and then all our tests were run. Concluding our test suite includes many unit, several functional tests and some integration tests, which result in a test coverage of 83.98% of code statements.

6.2 Baseline

To evaluate the scalability of our design we measure the latency of concurrent P2P connection establishments in both centralized and distributed GCS, for an ever increasing number of connect protocol instances.

We have developed a minified version of centralized GCS, as a baseline to compare our system against. Baseline system consists of three components

- a relay server,

- a caller application and
- a callee application

We describe the baseline system flow for the experiments. Both application types, once started, connect to the relay and send their identifier. After this, caller applications await for a UDP broadcast after which, they send a message to the relay requesting connection establishment. Each client calls its “respective” server application. For example, *client1* calls *server11* and *client2* calls *server12*. The minified connect protocol of baseline, consists solely of a client sending a *SignalingData* message to the relay, which in turn, forwards it to the intended recipient (which the message bears) and vice versa for server applications.

6.3 Setup & Strategy

For our experiments we have available a cluster of 11 nodes, connected via a single gigabit network switch. The server where we run baseline (relay), GCS and distributed GCS systems has an Intel Core i7-4820K processor at 3.70GHz and 64GB of memory. The rest of the ten nodes are classified into two groups. The first group consists of four servers which all have an Intel Xeon processor E5-2420 at 1.90GHz and 16GB of memory. The second group consists of six servers which all have an Intel Xeon processor E5-2620 at 2.10GHz and 32GB of memory.

The strategy for our experiments follows. As previously stated, we run baseline, GCS and Distributed GCS on the first server. An experiment instance for *N* concurrent connection establishments, provisions *N* callers in the group of six nodes and *N* callees in the group of four nodes. Note that, a “caller” and a “callee” consist of the pair of an application and its respective CMS.
For distributed GCS, the experiments strategy is the same. In this case we run three GCS instances in the server of the same node.

6.4 Connection Establishment

6.4.1 Generic Connectivity Service versus Baseline

In this experiment we measure the latency of connection establishment. For baseline, we measure the time from when a caller sends a request for P2P connection establishment to relay server, up until the channel is established.
For GCS, we keep two measurements, one from the perspective of applications and one from the perspective of daemons. The first one, starts from the request for connection establishment towards the CMS, up until receiving a successful response. It is important to note that this measurement includes also the time (after connection establishment), where the callee CMS inform the callee application, awaits for accept and then notifies the caller CMS through the P2P channel, which in turn, informs the caller application.
The second one, starts from the connection establishment request towards GCS, up until connection establishment. The reason for keeping two measurements is that the former is of interest for end user applications, while the second one provides a fair comparison against baseline system.

In figure 6.1, we demonstrate the experiment of connection establishment latency of the centralized version of our system against baseline. We start from 100 instances up to

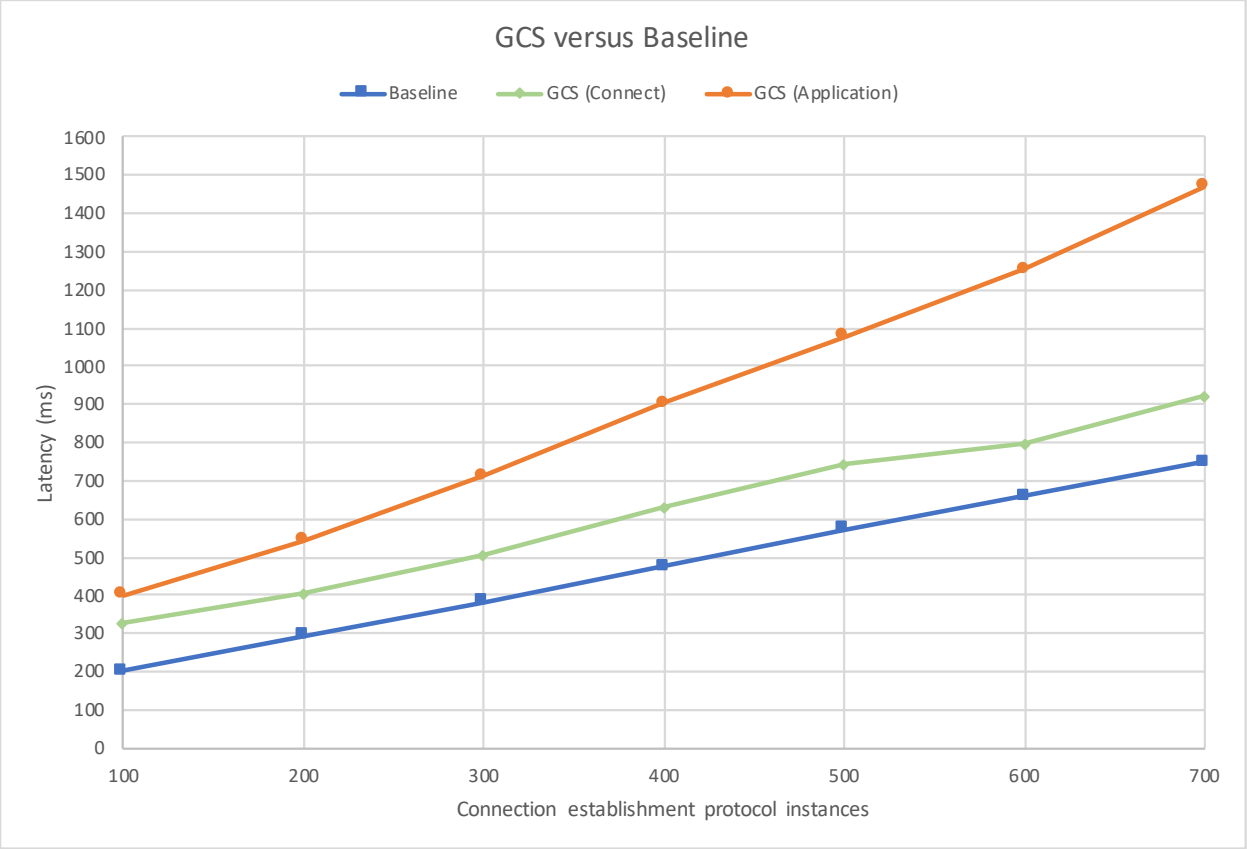

Figure 6.1: Connection Establishment latency (GCS, Baseline)

700 for three iterations (calculating the average for each instance). We notice that GCS (green line) is close to baseline, with an added latency of 140ms on average. This added latency can be accounted to the extra connect protocol phases among CMS and GCS, which baseline lacks. Specifically, the extra messages are `Connect`, `ConnectRequest`, `ConnectResponse` and `ConnectResult`, which entail four more hops, among caller CMS, GCS and callee CMS. As we described, baseline experiment solely relays `SignalingData` messages.

The reason we stopped at 700 experiment instances was because WebRTC library was crashing with no resources error in the lower-end servers we used.

Regarding the GCS latency from an application's perspective, we denote it includes the time from:

- the caller application requesting a connection towards its CMS (1 local hop)
- CMS initiating connect protocol, up until the channel is created
- callee CMS informing callee application of a new connection, which in turn accepts it (2 local hops)
- callee CMS informing caller CMS about the connection request outcome (1 hop)
- caller CMS informing caller application for connection establishment success (1 local hop)

Thus, the increase in latency, compared to the "GCS (Connect)" measurements (green line), can be accounted to the above overhead.

6.4.2 Generic Connectivity Service versus Distributed Generic Connectivity Service

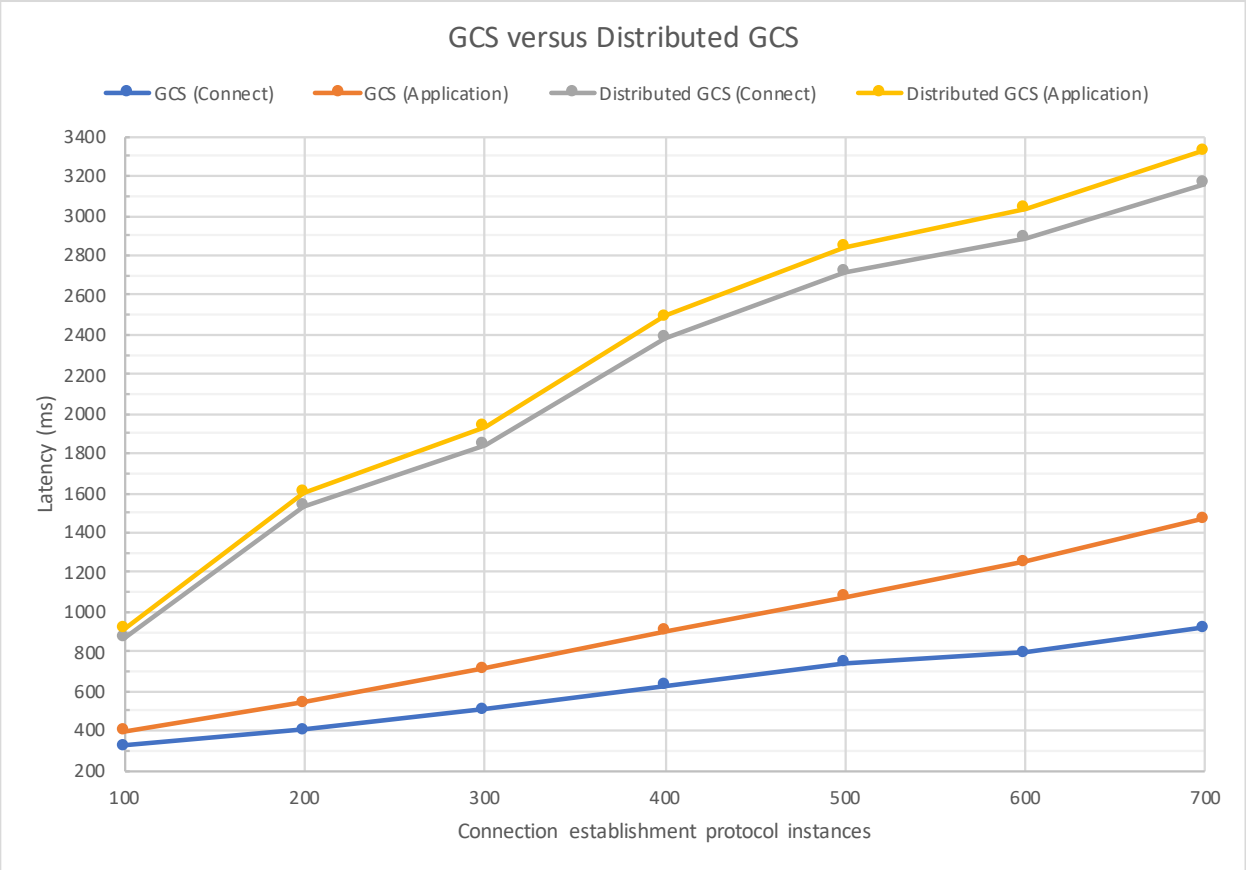

Figure 6.2: Connection Establishment latency (GCS, Distributed GCS)

In this experiment, we compare the two versions of our system, the one with centralized GCS and the distributed one, which comprises three GCS nodes (along with three DHT instances), running on the same node.

In 6.2 we notice latency increases by almost an order of magnitude in the distributed version. It is important to outline changes in the connect protocol, so that we can account for the extra latency and also identify areas of improvement.

Firstly, distributed connect protocol has two more inter-GCS hops for the connect protocol initial phase. That is, we have two extra messages, `RelayedConnectRequest` and `RelayedConnectResponse`. Apart from that, we also have an extra inter-GCS hop, for each signaling data message that is generated from each client. This is the `RelayedSignalingData` message of our protocol.

Regardless of the above, the most important factor for the added overhead, is that for each of the (relayed) messages above, i.e. the ones that have to go through the DHT, a node lookup (`FIND_NODES`) request is initiated.

Let's examine the case of delivering a signaling message for both centralized and distributed GCS. The centralized version has two hops, one from caller CMS to GCS and from GCS to callee CMS. The distributed version has these two hops and in addition, not only the node lookup process of the DHT, but also the actual relayed messages.

In addition, if a given connect session generates three signaling data messages, once these reach the caller's GCS, they will initiate three node lookup processes. We note that

we are not caching the results of FIND_NODES. This is an area of improvement, whereby we would cache these results with a sensible time-to-leave (TTL), which can be calculated dynamically and periodically according to the churn of the GCS network. Even a static value of a 1 minute TTL for cache entries would reduce the added overhead.

6.5 Channel Authentication

In this experiment we would like to demonstrate the latency added by the two way authentication protocol which can be run after the P2P channel has been established. Thus, we measure connection establishment time from a caller application perspective, for 100 instances (of the protocol), using three iterations, with and without channel authentication. In the case of channel authentication, the user's selected authentication method is her public key.

In table 6.1, we can see that the added latency is 43ms on average. We note that for one way of the prove identity protocol, the Prover cryptographically signs a nonce received by the Challenger, generating a ciphertext. Then, the Challenger cryptographically verifies the supplied ciphertext, along with Prover's public key and the generated nonce. The messages of this protocol flow through the newly established P2P channel. Concluding, the above process is repeated towards the other direction, i.e., Prover becomes a Challenger and vice versa.

Table 6.1: Channel Authentication latency

Channel authentication	Average (ms)	Minimum (ms)	Maximum (ms)
without	415	40	504
with	458	53	539

6.6 Handshake

In this experiment we measure the handshake protocol latency among our two systems. Distributed GCS comprises three GCS nodes (each running a DHT node), running on the same server, as the connect protocol experiment. Our experiment strategy follows. We start from 100 instances up to 700 for three iterations (calculating the average latency for each instance). All CMS systems use public key as authentication method, thus no communication towards BNS takes place.

To reiterate, handshake is a Challenge - Response protocol. The Challenger supplies the Prover with a nonce and the latter cryptographically signs it, providing a ciphertext. Finally, the Challenger verifies the ciphertext and the nonce using Prover's public key. This phase is identical in both versions of our systems.

The added overhead of the distributed system is finding the $handler$ node of each user. This entails that for each of the clients, GCS sends a FIND_NODES message through the DHT. This process retrieves the transport address and port of the DHT node whose identifier is closest to the identifier of a given user, namely the $handler$ node of the former user. For a CMS to connect to GCS, it also requires the port on which the GCS service is

listening to. Thus, after the FIND_NODES message, GCS sends a GET_PORT message over the DHT to the $handler$ node, to retrieve the port on which GCS service listens.

Having run the $FindHandlers$ process, if the client is to be handled by another GCS server, its connection is dropped and the CMS connects to its proper handler node. In sequence, the $handler$ node runs again the $FindHandlers$ protocol, so as to verify whether the current user should be handled or dropped. We chose to retrieve the nodes per user twice (once for the initial GCS server the user connects to and once for its $handler$ node) to ensure that regardless of GCS network changes, the user is handled by the appropriate node.

In the table 6.2 we can see the percentage of clients which initially connected to a node that wasn't their $handler$ node. Thus, had to repeat the FIND_NODES process, as described above.

Table 6.2: Percentage of clients not initially connected to a $handler$ node

Handshake instances	Percentage
100	67%
200	71%
300	73%
400	71%
500	74%
600	75 %
700	74%

In figure 6.3 we see the overhead incurred from running the distributed version of the handshake protocol, with an average of 72% sessions which did not initially connect to a $handler$ node.

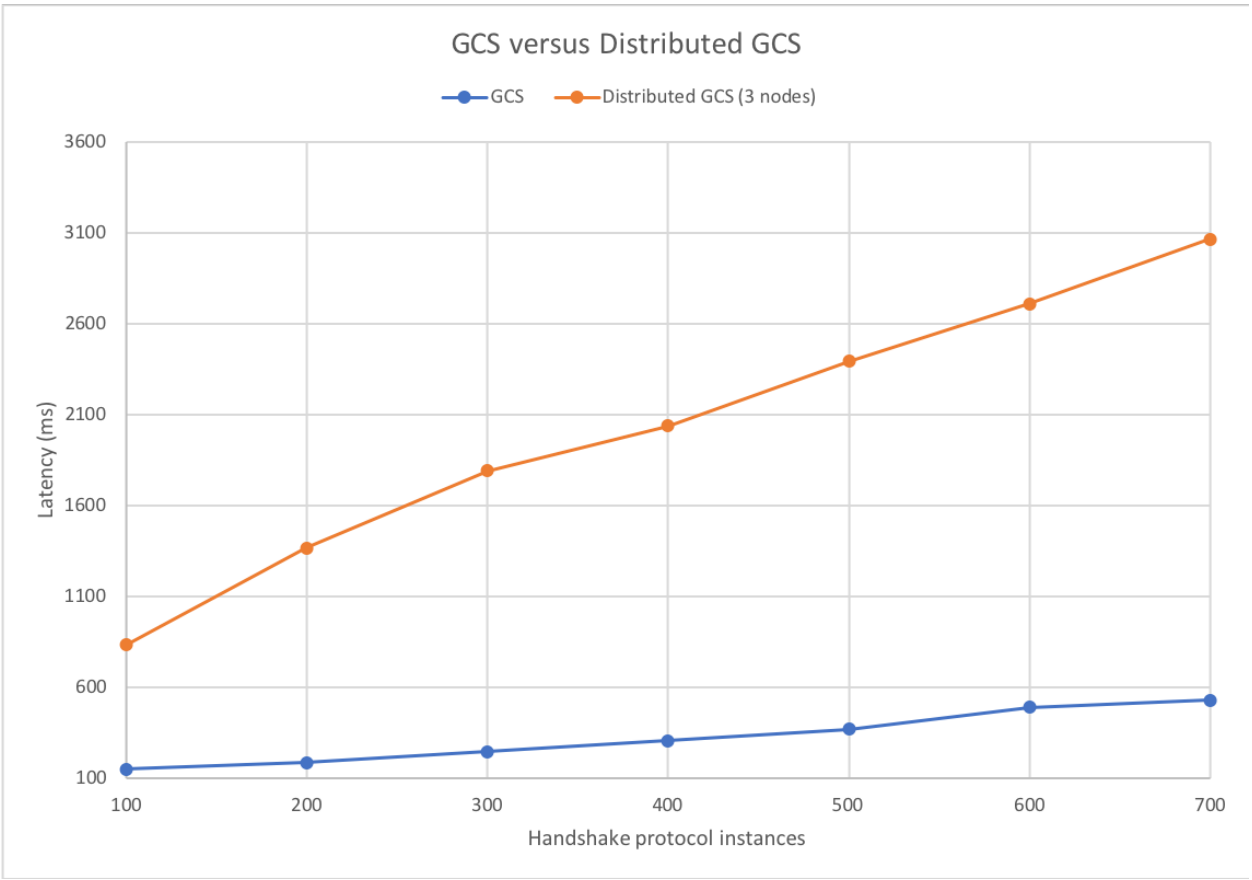

Figure 6.3: Handshake latency (GCS, Distributed GCS)

7. FUTURE WORK

We have several ideas on improving our system, both in terms of performance and in terms of user experience and usability. We elaborate on the most significant improvements below:

- Since we use Kademlia DHT as a routing layer, we could evaluate caching results of node lookup operations with an several eviction policy schemes. The simple one would be to evict an entry after a statically configurable time frame, e.g., a minute. A more elaborate scheme, would be to have a dynamically configurable time frame, which is adjusted periodically depending on the current churn rate of the GCS network.
- Our DHT nodes currently communicate using HyperText Transfer Protocol (over TCP). An improvement that would reduce the latency of the distributed version would be to develop a transport plugin for Kadence which uses solely TCP.
- Provide friends-list synchronization among user's devices. Currently, we assume the user manually maintains her friends-list updates from one device to another. In terms of user experience improvement, we could offer synchronization of said friends-list.
- Another area of improvement for user experience and usability, would to be provide a web interface as an administration panel of the local GCS daemon, whereby the user selects which applications can be reached from which of her friends.
- It would be interesting to incentivise users to offer a TURN/STUN server using a cryptocurrency, like StorJ [33] does for bandwidth. The idea is to easily allow anyone with a powerful internet connection operate a TURN server and measure their bandwidth offering for supporting the GCS network.
- Add replication to daemons connections with GCS network. Currently we allow only one GCS node per user. Ideally, as per Kademlia [24], there can be more than one nodes "handling" a specific key. This improves fault tolerance with quick failover after a GCS node departure. This entails handling user migration when a new node joins, as Kademlia does by replicating keys to newly joined nodes by periodically announcing keys. Thus a newly connected node that should handle a key, stores it and the older one deletes it when it expires.

8. CONCLUSIONS

We have designed and implemented a proof of concept of our system, a middleware platform facilitating peer-to-peer connections establishment among applications running on different users devices. The main novelty of our system, is that it integrates into its design the notion of a user with her devices, while related ones introduce only the notion of endpoints. It was our main goal to allow the user be the owner of her digital identity, which is why friends-lists are uploaded by the user at each new connection to the system and is deleted upon user's departure.
A benefit of this approach is that a user does not need to "rebuild" her friends-list from scratch for every new application she uses. Currently, the de facto approach is for the company providing an application to manage a separate friends list for each user and have knowledge of its contents from the get-go.

In addition to that, users may want privacy from other users (and to some extend the app provider), thus our system allows hiding a user's name by using public keys as identities and her friends-list by not providing the option to not even upload such list to our service.

We provided two architectures for our system, a centralized one and a distributed one. We opt for the distributed one, as it mitigates problems of single point of failure and trust, despite increased latency which can be improved in later iterations.

Our system abstracts away connection establishment, NAT traversal even behind firewalls by utilizing WebRTC, thus removes complexity from networked applications.

ABBREVIATIONS - ACRONYMS

ADT	Abstract Data Type
API	Application Programming Interface
BNS	BlockStack Naming Service
CI	Continuous Integration
CMS	Contact Management System
DHT	Distributed Hash Table
DNS	Domain Name System
EID	Endpoint Identifier
GCS	Generic Connectivity Service
HTTP	HyperText Transfer Protocol
ICE	Interactive Connectivity Establishment
IP	Internet Protocol
mDNS	Multicast Domain Name System
NAT	Network Address Translation
NFC	Near-Field Communication
PoW	Proof-of-Work
P2P	Peer-to-peer
QR	Quick Response
RTC	Real-Time Communications
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TTL	Time-to-leave
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VPN	Virtual Private Network
XMPP	Extensible Messaging and Presence Protocol
XOR	Exclusive Or

BIBLIOGRAPHY

[1] BitTorrent (Mainline DHT). https://en.wikipedia.org/wiki/Mainline_DHT. Accessed: 2019-11-17.

[2] Dropbox. https://www.dropbox.com/. Accessed: 2019-11-17.

[3] Flatbuffers. https://google.github.io/flatbuffers/. Accessed: 2019-11-10.

[4] GitLab. https://about.gitlab.com/. Accessed: 2019-11-10.

[5] Google drive: Free cloud storage for personal use. https://www.google.com/drive/. Accessed: 2019-11-17.

[6] Hangouts. https://hangouts.google.com/. Accessed: 2019-11-17.

[7] Kad network. https://en.wikipedia.org/wiki/Kad_network. Accessed: 2019-11-17.

[8] P2P Networks based on Kademlia DHT. https://en.bitcoinwiki.org/wiki/Kademlia#Networks. Accessed: 2019-11-17.

[9] Protocol buffers. https://developers.google.com/protocol-buffers. Accessed: 2019-11-10.

[10] RetroShare. https://retroshare.cc/. Accessed: 2019-11-17.

[11] WebRTC: Real-time Communication Between Browsers. https://w3c.github.io/webrtc-pc/. Accessed: 2019-10-28.

[12] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 181–194, 2016.

[13] Corey E Baker, Allen Starke, Tanisha G Hill-Jarrett, and Janise McNair. In vivo evaluation of the secure opportunistic schemes middleware using a delay tolerant social network. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2537–2542. IEEE, 2017.

[14] Juan Benet. IPFS-content addressed, versioned, P2P file system. *arXiv preprint arXiv:1407.3561*, 2014.

[15] Luca Deri and Richard Andrews. N2N: A layer two peer-to-peer VPN. In *IFIP International Conference on Autonomous Infrastructure, Management and Security*, pages 53–64. Springer, 2008.

[16] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 233–248. USENIX Association, 2006.

[17] Dead Canaries Foundation. Kadence distributed hash table. https://deadcanaries.gitlab.io/kadence. Accessed: 2019-10-28.

[18] Node.js Foundation. Node.js. https://nodejs.org. Accessed: 2019-10-28.

[19] Saikat Guha and Paul Francis. An end-middle-end approach to connection establishment. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 193–204. ACM, 2007.

[20] Hans Vatne Hansen, Vera Goebel, and Thomas Plagemann. DevCom: Device communities for user-friendly and trustworthy communication, sharing, and collaboration. *Computer Communications*, 85:14–27, 2016.

[21] Alan Johnston and Robert J Sparks. Session Description Protocol (SDP) Offer/Answer Examples. 2005.

[22] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). https://tools.ietf.org/html/rfc5766, 2010. Accessed: 2019-10-28.

[23] Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. Traversal using relays around nat (TURN): Relay extensions to session traversal utilities for nat (STUN). 2010.

[24] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[25] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[26] Rao Naveed Bin Rais, Mariem Abdelmoula, Thierry Turletti, and Katia Obraczka. Naming for heterogeneous networks prone to episodic connectivity. In *2011 IEEE Wireless Communications and Networking Conference*, pages 1091–1096. IEEE, 2011.

[27] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[28] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). https://tools.ietf.org/html/rfc5389, Oct 2008. Accessed: 2019-10-28.

[29] Jonathan Rosenberg and Christer Holmberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. https://tools.ietf.org/html/rfc8445. Accessed: 2019-07-10.

[30] Charalampos Rotsos, Heidi Howard, David Sheets, Richard Mortier, Anil Madhavapeddy, Amir Chaudhry, and Jon Crowcroft. Lost in the edge: Finding your way with signposts. *Proc. USENIX FOCI*, 2013.
[31] Peter Saint-Andre. Extensible messaging and presence protocol (XMPP): Core. 2011.
[32] Pierre St Juste, Kyuho Jeong, Heungsik Eom, Corey Baker, and Renato Figueiredo. Tincan: User-defined P2P virtual network overlays for ad-hoc collaboration. *ICST Trans. on Collaborative Computing*, 14(2), 2014.
[33] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. 2014.
[34] Dan Wing, Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. Session traversal utilities for NAT (STUN). 2008.