ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Integrating DAGA into the cothority framework and using it to build a login service

Lucas Pires

School of Computer and Communication Sciences

Decentralized and Distributed Systems Lab (DEDIS)

Master Thesis Project

February 2019

Responsible	**Supervisor**	**Supervisor**
Prof. Bryan Ford	Ewa Syta	Linus Gasser
EPFL / DEDIS	Trinity College / DEDIS	EPFL / DEDIS

Contents

1 *Introduction*

Authentication, like every interaction involving parties with different roles and interests, can be seen trough the eyes and concerns of the different protagonists. In many situations, the verifying party (e.g. service provider) wants to hold their users accountable for their actions[1] and needs to be sure that access to the service is only granted to authorized people[2]. While being legitimate those concerns can be seen as somewhat incompatible (or competing) with the growing desire (or necessity) for **privacy** of the proving party (user).[3] Hence sometimes, depending on the scenario, we would like to have authentication schemes that are more privacy aware than (e.g.) the traditional identifier-challenge systems, i.e. schemes that are built with more flexible tools and that offer a better or fairer trade-off between the competing needs of both sides. **D**eniable **A**nonymous **G**roup **A**uthentication (DAGA) is one of such anonymous authentication protocols that was designed by Ewa Syta during her Ph.D. thesis[14][4] and was first described in [15][5]. It "allows a user to authenticate as an anonymous member of … a group defined by a list of public keys"[14], is decentralized and offers a set of security features that will be detailed later (see 2.2).

The present work consists in first integrating DAGA into the *cothority*[6] framework by creating a new DAGA authentication service that aims to be easily (re)usable, i.e. not tailored to a specific scenario and offering only vanilla DAGA authentication and context creation protocols. Then using the new DAGA cothority to build a proof of concept login service consisting in an OpenID Connect (OIDC) Identity Provider offering DAGA authentication as a service. This allows every OIDC aware clients (or said differently: everyone) to delegates their user authentication to the DAGA cothority by mean of a well proven and established standard.

[1] e.g. in order to take actions to prevent dishonest behavior against their policies (such as vandalism, unfair usage, multiple accounts etc.) or the law.

[2] e.g. an online newspaper might want to be sure that the user is a subscriber or has purchased a plan.

[3]

e.g. users might agree with the needs of the provider without wanting to blindly trust the said provider won't abuse the 'functionality' and take it as an excuse to track, profile and sell their identified habits. or another class of users (whistleblowers or voting citizens or just anyone …) might disagree on the disclosure of any other information besides "I am legitimate" from the authentication process.

[4] Ewa Syta. *Identity Management through Privacy-Preserving Authentication.* PhD thesis, Yale, December 2015

[5] Ewa Syta, Benjamin Peterson, David Isaac Wolinsky, Michael Fischer, and Bryan Ford. Deniable Anonymous Group Authentication. Technical Report 1486, Department of Computer Science, Yale University, February 2014

[6] see 3.3.2

2 Background

In this chapter we will introduce various notions of interest such as authentication and identification before offering an introduction to DAGA.

2.1 Authentication, Identification and Privacy

Informally authentication is the act or process allowing a verifying party to ensure that a claim coming from the other party is true i.e. the other party is indeed who or what it claims to be, while identification results in disclosing or verifying an (often offline) identity. The following section aims at clarifying those notions. Readers interested in more complete and extensive definitions and discussions on these complex and related issues can see[14, p. 13-28] and[10][1] who are addressing the subject in great depth and where used to offer the following overview.

[1] Andrew Lindell. Anonymous Authentication. *Journal of Privacy and Confidentiality*, 2(2):24, 2011. doi: 10.29012/jpc.v2i2.590

Problem

Traditionally, authentication mechanisms whose purpose is to verify membership in a group (e.g. subscribers, users, moderators, citizens etc.) are often strongly linked with identification mechanisms, at least it is the case for the average John Doe experience who, e.g., was usually asked various information at enrollment time before being asked for its username or email (identification purpose) and password or challenge (authentication purpose) every time he wants to access the service. [2] *Authentication* mechanisms are necessary when we want to implement *Authorization* mechanisms, indeed we need to verify the trustworthiness of someone or something's answer to the question "why should we consider your request ?" before actually starting to wonder if we want to grant access to a resource or service. On the other hand, there are lots of use case where *identification* is not needed at all, might be undesirable or even dangerous. For instance some services might require identification at enrollment time but not at authentication time[3] or sometimes identification is only there as a poor attempt " to solve problems orthogonal to authentication and authorization, such as spam or misbehavior "[14], or for not any particularly good reason at all (legacy, "simplicity", laziness). That is why, as already mentioned in the introduction, we

[2] in such systems, the membership is decided "by (first) identifying an individual, then verifying that the individual is a member"[12] of the group.

[3] e.g. in voting systems, you usually have to prove your identity in order to request a ballot but the vote itself is anonymous.

need tools that allow service providers to build their identity managements systems in ways that take more into account the privacy of their end user and are fine-tuned to the specific needs of the application, systems where the trade-off between the competing interests of both side (user privacy vs provider confidence) is more balanced and sound. Because privacy has always been important[4] and is probably more now we live in the era of big data and observe privacy breaches everywhere.

[4] Privacy preserves our "right to be let alone"[20] and personal autonomy, while protecting us from targeted manipulation and self-censorship/inhibition.

2.1.1 *Definitions*

To lift the veil on the seemingly paradoxical notions of anonymity and privacy when discussing authentication, we need to introduce more formal definitions of the terms and processes being discussed.

Authentication (and obviously identification) is closely related to the management of identities. An identity need first to be established at enrollment time before being verified at authentication time and consists informally of an "ever-evolving[5] set of information about the client [/entity[6]] the identity belongs to"[14].

More formally an *identity* is defined to be a set of attributes[7] that specify a *group*[8] containing a single entity.[14] We can see that those definitions allow us to make the distinction between attributes sets that describe a single entity (identity) and attributes sets that describes multiple entities at once (group identity), and this key distinction allows us to offer a more formal definition of *identification* and *authentication*. Thus Authentication becomes the process allowing a verifier to confirm that an entity (prover) is a member of a group, that is the process verifying that the entity possesses all the attributes of the group identity in question. Whereas identification become a special case of authentication where the group is a singleton. Hence the authentication process only needs to verify that entities possess attributes of interest to the verifying side and this set of attributes may describe a sufficiently large group to yields a satisfying amount of privacy to the proving side. [9]

[5] Indeed every authenticated action can enrich the initial set of information (from enrollment time) a provider knows about a client.

[6] An **entity** represents a user, object or resource and can have multiple identities

[7] Attributes are properties of an entity, and a set of attributes, called a *group identity*, specifies a *group* "consisting of exactly those entities satisfying all attributes in the group identity"[14].

[8] set of entities in an *universe*, where an universe is a set of all entities of interests for a specific application)

[9] concept somewhat related to k-anonymity[13]

2.2 *Deniable Anonymous Group Authentication (DAGA)*

Following the previously introduced definitions, this section summarize chapter 4 of [14] to offer a tour of DAGA. For a more complete and formal description please refer to the original material.

DAGA is an *authentication* protocol run between an *entity* (user) at the proving side and a set of *anytrust*[10] servers at the verifying side. The protocol allows the servers to verify that an entity is indeed a member of a *group* specified by a list of public keys[11], while preserving the entity's anonymity in a forward-secure way. Moreover, authentication attempts are divided into epochs (or authentication rounds) and each successful authentication results in a unique (per-round) tag that allows linking authentications of the same en-

[10] the servers are "collectively but not individually trusted"[14], that is we assume there is at least one honest server.

[11] accordingly, an entity belongs to the group if it possesses/knows one of the corresponding private keys.

tity thus offering to the verifying side the proportionality[12] property it sometimes expects/needs.

[12] see below 2.2.1

2.2.1 *Properties*[13]

[13] see[14, Sect. 4.6] for complete and formal descriptions, assumptions and proofs.

DAGA offers the following properties.

Completeness
: Entities that are member of the group and following correctly the protocol authenticate successfully[14]

[14] Unless the process is aborted upon exposure of misbehaving or dishonest server(s). If nothing is done administratively to change the situation then the dishonest server(s) can prevent any progress.

Soundness
: Only entities that are member of the group can authenticate.

Anonymity
: No PPT[15] adversary can guess which member authenticated with probability bigger than random guessing.

[15] Probabilistic Polynomial Time, probabilistic (non deterministic) adversaries/algorithms/turing machines that are polynomially bounded (in time).

Forward Anonymity
: Once an authentication round (epoch) is ended, it is impossible to break the anonymity of any entity, even if the private keys of nearly all actors are compromised[16]

[16] In its basic (and implemented) form, DAGA requires that the key of the honest server remains undisclosed (but this requirement can be relaxed).

Proportionality
: Authentication of an entity results in same final linkage tag for each authentications made during same epoch. (but authentications of same entity made during different epochs are unlinkable). The linkage tag act as anonymous or pseudonymous IDs of the entity during a round.

Deniability
: The entity can successfully deny having ever attempted to authenticate itself, in fact it is impossible to prove that any entity authenticated at all.[17]

[17] The entity proves her membership (authenticates) by using an interactive zero-knowledge proof of knowledge, meaning that the interaction transcript cannot convince anyone but the original actors and is indistinguishable from a transcript obtained by running a simulator.

Forward Deniability
: The deniability property is retained even if an attacker gains additional knowledge of the private keys of all entities/users.

2.2.2 *Overall description*[18]

[18] see[14, Sect. 4.3] for complete description.

Authentication Context

As already mentioned, DAGA divides authentication attempts into epochs or authentication rounds. A client wanting to authenticate itself as a member of a group during an epoch needs such a group description as well as other round specific information which are packed in a public structure called an *authentication context*.

An authentication context C contains:

$\vec{\boldsymbol{X}}$*:* the long-term public keys of the n entities/users[19]

[19] The keys used are Diffie-Hellman-Merkle (DH) key pairs.

$\vec{\boldsymbol{Y}}$*:* the long-term public keys of the m servers

$\vec{\boldsymbol{R}} = \{\boldsymbol{R_1}, \ldots, \boldsymbol{R_m}\}$*:* each server's commitment $R_i = g^{r_i}$ to its per-round secret r_i

$\vec{\boldsymbol{H}} = \{\boldsymbol{h_1}, \ldots, \boldsymbol{h_n}\}$*:* the unique per-round generators of $\mathcal{G}$ associated to each clients[20]

$\boldsymbol{\mathcal{G}}$ *and* $\boldsymbol{g}$*:* implicitly and obviously all actors know the algebraic group $\mathcal{G}$[21] in usage and one common generator g. (the same used in the above descriptions in multiplicative notation)

[20] "such that no one knows the logarithmic relationship between any h_i and g or between h_i and h'_i for any pair of clients $i \neq i'$"[14]

[21] see subsection 3.3.1

Authentication

Then, once in possession of the context, the i^{th} entity can start its authentication process which consists in building an initial linkage tag $T_0 = h_i^s$ using its per-round generator[22] as well as proving its membership and the correctness of its computation to the verifying servers.

The servers are convinced by an interactive *HVZK*[23] proof of knowledge[24] for the following "OR"-predicate :

$$PK\{(x_i, s) : \vee_{k=1}^{n}(\text{"}knows\ k^{th}\ secret\ key\text{"} \wedge \text{"}T_0\ correct\text{"})\}$$

Once done, the entity embeds the resulting tag and proof transcript in its authentication message M_0 and send it to an arbitrary server. Then all the servers will, in turn (ring order based on indices in context), process the request, by each :

1. verifying the client's proof and all the previous (if not first) server's proof (aborting the process if either of them is invalid)
2. scrubbing their shared secret s_j from the tag being built and replacing it by their per-round secret r_j to build an intermediate tag $T_j = (T_{j-1})^{(r_j)(s_j^{-1})}$
3. adding a proof stating that they did their work correctly or exposing a misbehaving client (in that case they set $T_j = 0$)

At the end " all servers learn a final linkage tag $T_f = h_i^{\prod_{k=1}^{m} r_k}$ "[14] in case of success or $T_f = 0$ in case of failure. Obviously communication between all actors need to take place over secure authenticated channels and if we don't want to destroy the purpose of DAGA we need an anonymous routing mechanism[10].

[22] $s = \prod_{j=1}^{m} s_j$, where $sj = H(Y_j^{z_i})$ is a shared secret (with server j) derived from DH key exchange using a new ephemeral DH key $Z_i = g^{z_i}$ and a suitable hash function H.

[23] Honest-Verifier Zero-Knowledge, see later 3.3.2

[24] designed and executed following techniques described by Camenish and Stadler in[5]

3 Implementation

The following chapter will present the work that was done during the project the design choices and the current state of advancement. We begin by describing the goals and current state before detailing the work done on the three main parts of the project individually:

1. the DAGA library (see subsection 3.3.1)
2. the DAGA cothority service (see subsection 3.3.2)
3. the login service and the proof of concept (see subsection 3.3.3)

The first part uses the work from a previous student as a starting basis[19][1], the second part builds on the previous to integrate DAGA into the cothority framework, while the last part makes use of the previous parts and CoreOS's dex project[1][2] to implement a login service.

[1] Arthur Villard. report_pfs_pop.pdf. Technical report, 2017. URL `https://github.com/dedis/student_17/tree/master/pfs_pop`

[2] Announcing dex, an Open Source OpenID Connect Identity Provider from CoreOS | CoreOS. URL `https://coreos.com/blog/announcing-dex.html`

3.1 Overview and goals

As already mentioned in the introduction[3], the final goal of the project is to offer *DAGA authentication as a service*[4]. We aim at letting current and future system designers leverage easily DAGA's promises in their design and thus offering them the opportunity to take more into account the privacy of their end-users whether it is critical for their mission or not. As hinted we would like our solution to remain as generic as possible, i.e. not tailored and tied to particular uses cases and deployment scenarios. Naturally we will need to keep the properties[5] of DAGA while doing so and we will try to separate as much as possible the different components and building blocks into well defined abstraction levels.[6] This implies too that we would like to not pollute the created lower level primitives (e.g. DAGA cothority) with the requirements of the upper ones (e.g. login service) while implementing them, and thus offering them untouched for other future usages.

[3] chapter 1

[4] or a "login service" as put in the title

[5] see subsection 2.2.1

[6] e.g. to build the "login service", everything from the DAGA library to the full cothority service, looks just like a "simple" authentication primitive:
$yes; tag | no; 0 = daga(request)$

3.2 Current state and contributions

As part of the current work the following contributions were made:

- the DAGA library was ported to kyber.V2, the client-side code was rewritten, issues have been identified and some fixed, documentation as well as guidelines for the rewriting of the server-side code have been added.

- the library was used to implement DAGA in the cothority framework, featuring:
  - a new *service* allowing to build cothorities supporting DAGA authentication and the creation of DAGA contexts
  - three new *protocols* to support the service
  - a CLI app to interact with it
  - test coverage edging 80% for the protocols and service
  - boilerplate allowing to run simulations locally as well as on DETERLab's testbed facilities
  - compliance with the tool that generate proto files to allow future interop with other languages
- a system allowing 3rd-party services to delegate the authentication of their users (through the well established OpenID connect standard) to a running DAGA cothority was implemented.
- working, reproducible and customizable proof of concept of the system mentioned above

3.3 Building blocks / Components

As said, the project can be divided into three main and reasonably well separated level of implementation, or components. This section will describe them in turn.

3.3.1 DAGA library

This part builds upon the work of a previous student available on Github.[18][7] The library is implemented in golang[11][8] and uses heavily the Kyber cryptography library developed at EPFL by the DEDIS team[7][9].

[7] Arthur Villard. Deniable Anonymous Group Authentication, 2017. URL `https://github.com/dedis/student_17_pop_fs`

[8] Rob Pike, Ken Thompson, and Robert Griesemer. The Go Programming Language. URL `https://golang.org/`

[9] DEDIS. Advanced Crypto Library for Go, . URL `https://github.com/dedis/kyber`

The goal of this part of the project was to ultimately integrate the new DAGA library into Kyber. The previous work was first ported from kyber.v0 (Crypto.v0) to kyber.v2, before being reworked to address some issues.

Since the goal was to integrate the library into Kyber and that one of the main point of Kyber is to " facilitate upgrading applications to new cryptographic algorithms or switching to alternative algorithms for experimentation purposes"[9] we decided to try to leverage as much as possible of the facilities offered by Kyber.

The first step in this direction was the addition of a new Suite[10] interface for DAGA, that will notably allow future users to switch to other cryptographic building blocks if needed or wanted. For instance the original paper worked in a Schnorr group[11] (i.e. a large

[10] A Suite is " an abstract interface to a full suite of (...) crypto primitives chosen to be suited to each other and have matching security parameters. " quoted from

[11] specifically, the group of quadratic residues modulo a safe prime.

prime-order subgroup of the multiplicative group of integer modulo p where p is prime), while the previous and current work implementation works with a prime-order group constructed with the twisted Edwards curve birationally equivalent to Curve25519[2](i.e. the suite uses the same group that is used in Ed25519's variant of the EdDSA signature scheme).

We can see that from DAGA's perspective either implementation is fine as long as it satisfy DAGA's assumptions on the algebraic group being used, which is the case.[12] But from an engineering perspective this means we need to identify the individual high-level operations that are needed by DAGA and that would need to be dealt with different level of care depending on the underlying group and primitives used and their implementation in Kyber. (e.g. for diffie-hellman key exchange, we usually need to validate the public keys.[16][13] but with current EC concrete implementation we, at least in theory, don't need to care)[14]

[12] DAGA is heavily based on discrete logarithm cryptography, thus DAGA assumes a group $\mathcal{G}$ where the decision diffie-hellman problem[3] is assumed to be hard (DDH assumption).

[13] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society. ISBN 978-1-891562-46-4. doi: 10.14722/ndss.2017.23171. URL `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/measuring-small-subgroup-attacks-against-diffi`

[14] In the current curve if the secret keys are correctly generated to be multiple of 8 (the curve's cofactor) then we don't have to check if the remote public key is in the correct subgroup, because even if it is not, the DH shared secret cannot leak (by construction) any bits of the targeted secret key. However in practise it is not the case, we noticed that the code generating the secret keys in Kyber is, at the time of writing, not correct, still seems that it is not a big deal because I am under impression that bad Points are rejected / not representable by the library.

As a second major step we can mention, the rewriting of the client-side code and notably the PKClient proof[15] related code to make use of the existing kyber.Proof framework instead of rewriting whole proof machinery by hand.

[15] see section 2.2.2

This was done for multiple reasons, readability and maintainability for a start, it lessen the bug surface, i.e. we only need to look for proof related code at one well written, tested and documented place (the kyber.Proof package). Then we can view the same rationale from another perspective, since the proof framework is part of Kyber and we eventually want to integrate DAGA into Kyber, we might give a try at the "eat your own dog food" motto and use it in our own projects. To expand on this topic, we had to design some API wrappers around the proof framework to interface with it in a more easy and familiar way.[16] Those wrappers could maybe be adapted into an additional reusable API/pattern for the proof framework to ease future usage where, like in our case, the user needs to build interactive proofs across different machines.[17]

[16] see the clientProverCtx, clientVerifierCtx and their methods in client_proof.go.

[17] they allow to synchronize easily with a running proof.Prover, to extract the commitments in order to forward them elsewhere, then to continue the proof once a challenge was received from the remote end, etc.

Finally we introduced more documentation with links to original DAGA paper, better error messages and minor refactoring to make the code more readable DRY and idiomatic.

Still in our opinion the server side code is not production ready, nor of same quality level as the remaining of Kyber and thus should be mostly rewritten too (including the server proofs). This has not been done because the implementation was good enough for our current goal and we moved on the next phases in order to not lose more time on it. The library is well tested but the tests are mostly unmaintainable and it would be easy to miss some important cases. They were refactored to introduce the proper usage of testify and to fix some easy things but they were ultimately left in the state they were found. This leaves the library in an unfinished state and for all of these reasons and the fact that I don't fully endorse it in its current state the integration in Kyber was not performed.[18] Still the

[18] at some point after the port to kyber.v2 a WIP pull request was made and ultimately closed for the same reasons.

code is full of comments detailing the direction to take to resolve the situation.

3.3.2 DAGA Cothority

This part uses the previously introduced library to implement DAGA in the cothority framework[9][19].
Cothority is a DEDIS project that

> provides a framework for development, analysis, and deployment of decentralized, distributed (cryptographic) protocols. A given set of servers running these protocols is referred to as a collective authority or cothority. Individual servers are called cothority servers or conodes[19] .

[19] DEDIS. Scalable collective authority., . URL `https://github.com/dedis/cothority`

The following figure depicts an high-level overview of the new DAGA cothority that will be subsequently described.

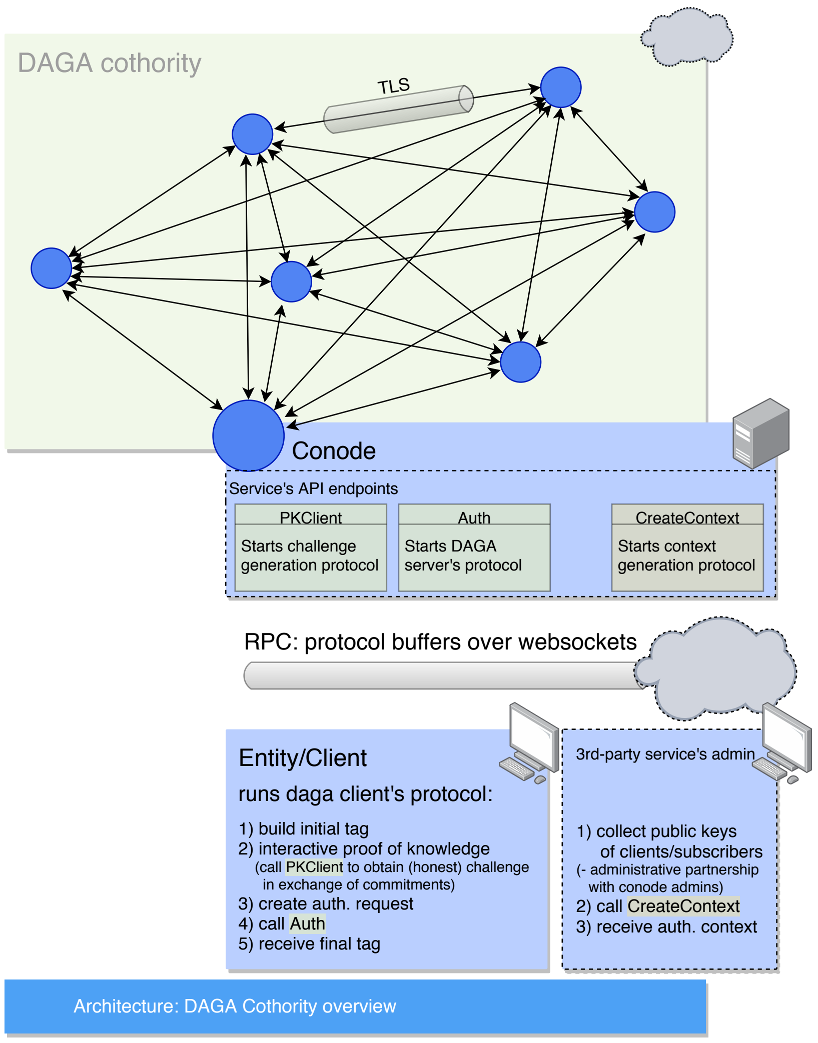

DAGA cothority
TLS
Conode
Service's API endpoints
PKClient
Starts challenge generation protocol
Auth
Starts DAGA server's protocol
CreateContext
Starts context generation protocol
RPC: protocol buffers over websockets
Entity/Client
runs daga client's protocol:
1) build initial tag
2) interactive proof of knowledge
(call PKClient to obtain (honest) challenge in exchange of commitments)
3) create auth. request
4) call Auth
5) receive final tag
3rd-party service's admin
1) collect public keys of clients/subscribers
(- administrative partnership with conode admins)
2) call CreateContext
3) receive auth. context
Architecture: DAGA Cothority overview

Figure 3.1: high level overview of DAGA in the cothority framework

Context Creation

As said in the background chapter, every party needs to have access to a public (in the "not sensitive" sense) authentication context[20] before being able to do anything. To create such context, the protocol described in [14, Sect. 4.7.3] was implemented.

[20] see section 2.2.2

A client can (typically an admin of a 3rd-party service) create a context by:

1. Gathering the long-term public keys[21] of the *group* members (or entities, typically the subscribers of the service).
2. building a cothority by establishing individually and administratively partnerships with the admins that run DAGA enabled conodes. (but we can imagine "open access" conodes) and creating a roster[22] out of the process.
3. finally sending an authenticated CreateContext request (containing the keys and roster) to a random conode in the roster.

[21] To do so it is envisioned to offer tools allowing to scrap them out of a POP[4] party transcript for instance, but the admin is free to collect them by any mean that fits its agenda.

[22] a structure describing basically how to reach them and communicate securely with them (public keys)

Then the conode receiving the request checks first if it has a matching partnership before initiating a new instance of the context generation protocol, with the other nodes listed in the roster as partners. During the protocol run, all other conodes will check in their turn if they accept the request before acceding to the request of the leader and helping it building the context. At the end of a successful protocol run (where nothing strange has been detected by any node), the new context is sent back to the client and every node will start serving the newly created DAGA context.[23]

[23] or said differently, accepting authentication requests under the context, and hence establishing effectively a new *collective authority* for the context by doing so.

<u>Current state:</u>

As currently implemented, each conode can serve multiple different contexts at once (each with possibly different daga server identities/public keys) and at the end of each context creation the state is persisted in a bbolt database, allowing a node to retrieve it after a potential shutdown or crash without destroying permanently the cothority and/or forcing everyone to rebuild a context.

However there are currently no facilities/boilerplate to manage and conduct the administrative partnerships that was mentioned in the described scenario nor to authenticate the 3rd-party services admins, hence currently the conodes are "open access" DAGA nodes. Still the service and protocol were implemented while keeping those ideas in mind and already have places to check such things, making the addition straightforward.[24] One of the ideas would be, interestingly, to reuse DAGA for that purpose, we can imagine that we define administratively/offline a context whose members are the people that have a partnership with the conode that allow them to create contexts (similar to authenticated darcs). Then a running cothority (serving the context) and containing the conode, authenticates the request.

[24] Have a look at the dagacothority/service/service.go

As seen the DAGA cothority was designed such that every conode retains completely its free-will, each conode is independent and tied only by the contexts it serves to form possibly multiple cothorities with possibly different partners. Finally quoting [14] those DAGA enabled conodes are expected to be

> deployed by a federation of organizations wishing to support responsible forms of anonymous participation (...), anonymity system providers such as the Tor project, non-profit organizations whose aim is to further online privacy and anonymity, or even for-profit organization desiring strong guarantees and large anonymity sets for their clients.

Authentication and Σ-Protocol

A client having in possession a suitable authentication context can authenticate itself to the DAGA cothority by sending its authentication message to the *Auth* endpoint of a random conode.

As already described in section 2.2.2, to build such an authentication message, the client has to conduct a zero-knowledge proof of knowledge with the cothority at the verifying side. To do so it will call the *PKClient* endpoint of a random conode to request a challenge in exchange of commitments.

Since the proof in question is of the Σ-protocol kind, the zero-knowledge property[25] holds only if the verifiers (thus the challenge) are honest[26]. Hence we need to have the servers " collectively generate c_s [(the challenge)] so that each server, which would include at least one honest server, contributes its randomness towards c_s. "[14] Moreover if the prover can predict the challenge (maybe with the help of some dishonest servers), it will allow it to cheat by crafting its first move (the commitments) using the verification formula or put differently, by abusing the HVZK proof structure and using the "simulator + rewind verifier" approach.

[25] and hence the anonymity and deniability properties of DAGA

[26] (special HVZK property)

To summarize the distributed challenge generation protocol has two main purposes:

1. the obvious one : make the "verifier"(cothority) honest to keep the proof zero-knowledge. this is solved by the fact that we assume an anytrust setting, at least one honest server will contribute its randomness to the challenge.
2. the less obvious one: convince individually the servers that the challenge is honest (they trust themselves to add their randomness to it) and that the client and other servers cannot collude and cheat.

Hence at the end both the honest client and the honest server are ensured of the randomness and "honesty" of the cothority as a whole.

Additionally, the proof being interactive[27], it convinces only the participants. As a consequence, if the servers want to verify it later (Auth endpoint) based only on the transcript embedded in the authentication message[28], additional care and mechanisms should be put in place. Indeed if nothing is done (as it was the case with the

[27] in order to offer the deniability property of DAGA

[28] That's how DAGA is described in [14],

previous implementation[18]) the prover can cheat using the same technique described above.

There are two approaches to prevent this from happening:

1. keep state (commitments and challenge) between the *PKClient* and *Auth* requests at each node, and rely only on it to verify the proof. This approach was chosen by [21][29]
2. or avoid by following the "keep state in clients" motto of RESTFul services.

We chose the second option, and implemented it by having the servers tie the challenge to the commitments by each issuing a signature on *challenge*||*commitment* instead of only challenge at the *PKClient* step. This way, when the client calls later the *Auth* endpoint, the servers can individually verify that the challenge and commitments have not been altered by verifying their own signatures[30].

[29] David Isaac Wolinsy. DAGA as a service., 2014. URL `https://github.com/davidiw/dagas`

[30] the signature ties the challenge to the commitments and at the same time protects authenticity and integrity of the pair

3.3.3 Login service

This part will describe the designing of the login service's architecture.

Intro, problems, options

Up to now we have a running DAGA cothority service that can authenticate clients.[31] Now we want to introduce a 3rd actor in the picture, a service provider that want to use the running DAGA cothority to authenticate its end-users on its behalf. We already implemented facilities to create authentication contexts out of a list of public-keys, remains now to introduce a communication protocol between the three actors that allow for such delegation while preserving DAGA properties.[32]

[31] see Figure 3.3 below.

[32] notably deniability, which as you will see is not that easy.

Figure 3.2 shows a typical authentication delegation flow and introduce the classical terminology that will be used throughout the chapter.[33]

[33] it was adapted from a figure present in a really good blog post [17]

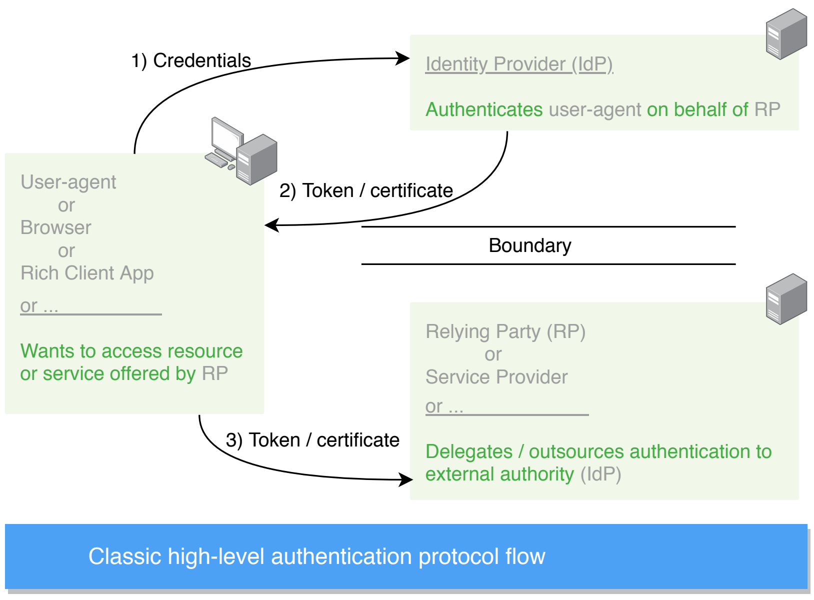

1) Credentials
Identity Provider (IdP)
Authenticates user-agent on behalf of RP
User-agent
or
Browser
or
Rich Client App
or ...
Wants to access resource or service offered by RP
2) Token / certificate
Boundary
Relying Party (RP)
or
Service Provider
or ...
Delegates / outsources authentication to external authority (IdP)
3) Token / certificate
Classic high-level authentication protocol flow

Figure 3.2: Classic high-level authentication protocol flow

Authentication delegation mechanisms are necessary since in all complex systems, we cannot ask every players that provide access to services or resources to deal with the management of identities and authentication as well.

Additionally, while it is true that in theory DAGA can be used anywhere as an ideal replacement of linkable ring signatures (LRS) schemes with the added benefit of deniability and forward security[14], the replacement is not straightforward in practise. In the LRS case everyone can verify the signature whereas in vanilla DAGA only the cothority is convinced of the user authentication[34] and the result of the process is only a linkage tag / pseudonymousID that is to be considered public since all the servers have access to it.

[34] and only at the time of verification if there are no other session mechanisms in the picture

The only remaining option, as hinted, would be to require that services interested into using DAGA as authentication mechanism be part of the DAGA cothority. Not only this is not practical for complex systems, but we would need to either modify the current DAGA cothority code to accommodate for such scenarios (e.g. provide hooks etc.) or tolerate having specialized version of DAGA nodes for each application. Finally this would imply that we are mostly done since the remaining work would be to implement such tight integration mechanisms, which is meaningless without having well defined specifications for the services requirements.

Hence if we want to be as generic and simple as possible and not restrict ourselves with such specific design, we need to think of ways allowing to delegate authentication, i.e. ways allowing the user-agent to authenticate to the 3rd-party with the help of the DAGA cothority.

Strawman designs, options

To visually represent what has been said above and develop further the options facing us, we start from the following situation:

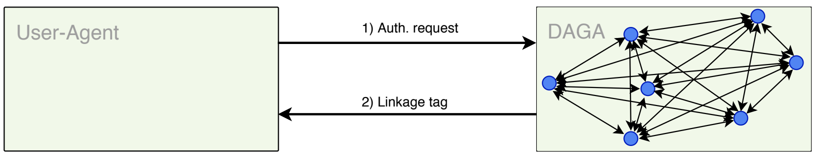


Figure 3.3: basic no delegation scenario

As a first way to address the problem mentioned above, we can do the following (for instance):

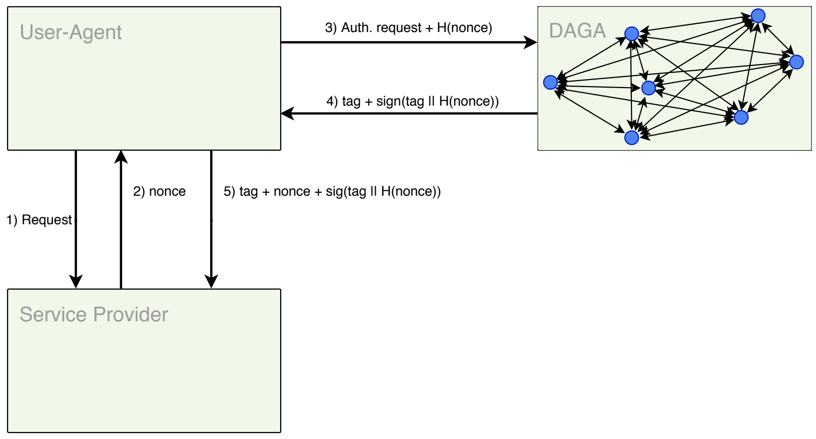

User-Agent
3) Auth. request + H(nonce)
4) tag + sign(tag II H(nonce))
DAGA
1) Request
2) nonce
5) tag + nonce + sig(tag II H(nonce))
Service Provider

Figure 3.4: adds a way to convince 3rd-party that user-agent authenticated with the DAGA cothority

But now we just modified the DAGA cothority to make it handle nonces and conversely this requires to closely tie the 3rd-party services sign-in code to the DAGA case. Another hidden assumption here is that the 3rd-party service has access to the user assets, effectively we put it in a position where it can authenticate itself as the user-agent to any authorization mechanism pipelined behind. Hence we just restricted ourselves to specific use cases s.t. forums or wikis.

To address the mentioned issues, we can do the following:

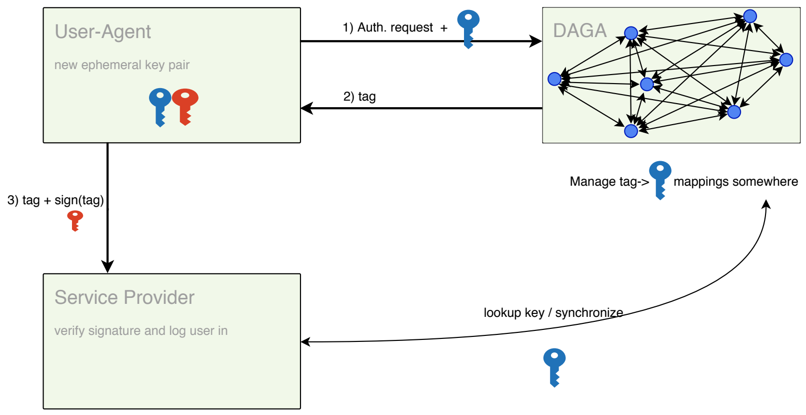


Figure 3.5: adds a better way to convince 3rd-party that user-agent authenticated with the DAGA cothority

This is essentially what was described in [14, Sect.4.5.1] and what was done by the dagas implementation[21][35]. With each successful authentication the user-agent has the opportunity to register with the cothority a new ephemeral public key that is not tied to its long-term ID / public key but to its linkage tag / pseudonymousID. Furthermore, now the service provider doesn't know the ephemeral secret key and thus cannot authenticate itself as the user-agent.

[35] David Isaac Wolinsy. DAGA as a service., 2014. URL `https://github.com/davidiw/dagas`

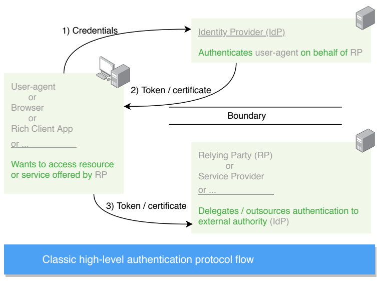
This is already better than the previous designs since now the cothority is almost not modified, we only need to introduce an additional "key server" role that can be implemented in different non-intrusive ways to the cothority implementation[36].

[36] (e.g. cothority can just publish *tag* $\mapsto$ *key* mappings on a public key-value store etc.)

However as a new downside we just lost forward-security and deniability, if client's ephemeral secret key is ever compromised we can retroactively compromise its anonymity in all past exchanges, since we are now convinced that only it could have issued these signatures. In fact looks like we just transformed DAGA into a LRS scheme offering the possibility to rotate keys as a bonus. To mitigate this issue we can use very short lived keys etc.. Additionally we would need to make sure too that any conode processing the request cannot replace the key with one under its control.

Up to now, all the described approaches are still somewhat home-cooked and not readily interoperable with existing systems deployed in production. If we want to go down that path and still interface easily with existing systems we would need to provide bridges or connectors, one of such option is to provide authentication strategies for well established authentication middleware. e.g. passport.js, accounts-ui or omniauth for respectively the node.js Meteor and Rails web app worlds.

The next and last high-level option we will describe essentially just looks like Fig. 3.2 (see in margin) that introduced authentication delegation.


1) Credentials
Identity Provider (IdP)
Authenticates user-agent on behalf of RP
User-agent
or
Browser
or
Rich Client App
or ...
Wants to access resource or service offered by RP
2) Token / certificate
Boundary
Relying Party (RP)
or
Service Provider
or ...
Delegates / outsources authentication to external authority (IdP)
3) Token / certificate
Classic high-level authentication protocol flow

In Fig. 3.6, instead of reinventing the wheel we rely on existing and well proven protocols and standards for authentication delega-

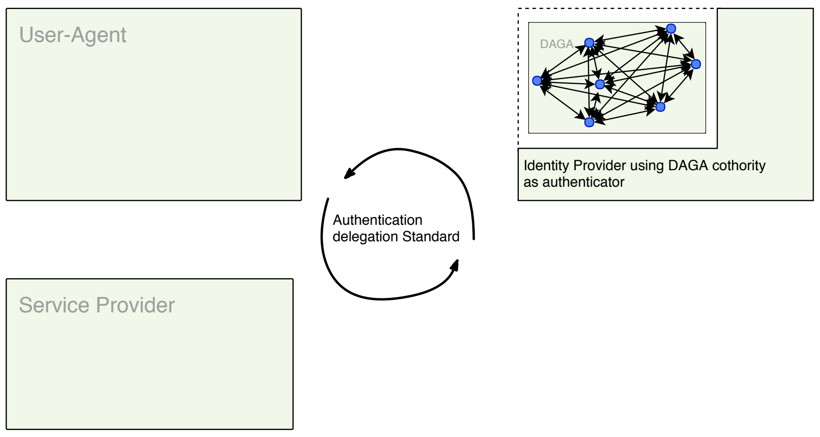

User-Agent
DAGA
Identity Provider using DAGA cothority as authenticator
Authentication delegation Standard
Service Provider

Figure 3.6: introduce new trustee entity (IdP) and uses standardized ways to convince 3rd-party that user-agent authenticated with the DAGA cothority

tion. It is the easiest way to have something working quickly and correctly[37] that would offer us the additional benefit to interface easily with existing services that most definitely already use those mechanisms in their authentication workflow. To do so we need to introduce a new actor in the picture, the IdP that will use the DAGA cothority as authentication primitive.

[37] designed with care and over time by people and industries that are experts in the field.

Chosen solution and PoC

Now that we discussed the importance to use an authentication delegation system and exposed different options, we need to choose how to implement such system and how it will be used in our PoC. From the decision keys introduced in the previous description, we chose to use the last option, and we fixed the authentication delegation protocol to be OpenID connect.

OpenID Connect[6][38] (OIDC) adds an identity layer to, and build upon, the well known OAuth 2.0 authorization framework. It is one of the major standard for authentication in the modern world. The other choices of framework could have been *SAML*[39] or eventually Kerberos in controlled environments.

[38] CoreOS. OpenID Connect. URL `https://openid.net/connect/`

[39] Security Assertion Markup Language

To build our IdP we chose to use the free-software dex OIDC identity provider which is written in Go and maintained by the CoreOS project.

Fig. 3.7 describes the system we designed to power the PoC.

The moves in blue are related to the DAGA authentication protocol, the moves in red depicts user input/action and the remaining moves in black are the move of the standard OIDC code flow,[40] where

[40] we can choose whichever OIDC flow we want, this is offered for free by OIDC and DEX.

1. the RP redirects the user to the IdP
2. the IdP authenticates the user (here using DAGA)
3. the IdP redirects back the user to the RP with an authorization code

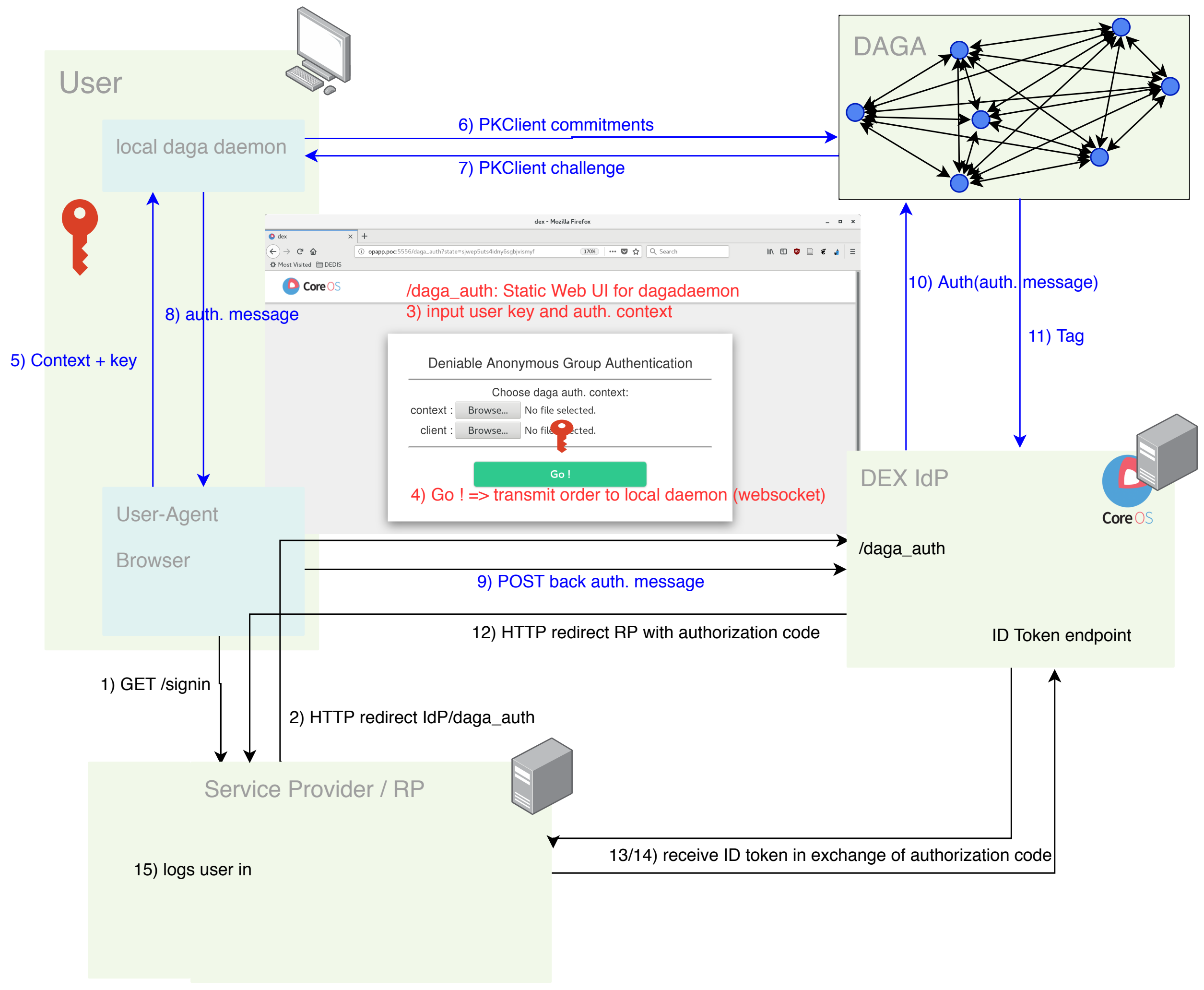

User
local daga daemon
DAGA
6) PKClient commitments
7) PKClient challenge
5) Context + key
8) auth. message
dex - Mozilla Firefox
/daga_auth: Static Web UI for dagadaemon
3) input user key and auth. context
Deniable Anonymous Group Authentication
Choose daga auth. context:
context : Browse... No file selected.
client : Browse... No file selected.
Go !
4) Go ! => transmit order to local daemon (websocket)
10) Auth(auth. message)
11) Tag
DEX IdP
Core OS
/daga_auth
User-Agent
Browser
9) POST back auth. message
12) HTTP redirect RP with authorization code
ID Token endpoint
1) GET /signin
2) HTTP redirect IdP/daga_auth
Service Provider / RP
15) logs user in
13/14) receive ID token in exchange of authorization code

Figure 3.7: The designed auth. delegation system, the figure shows the standard OIDC code flow as well as the moves related to the DAGA authentication protocol

4. the RP talks to the IdP to exchange the authorization code against an ID token which asserts the identity of the user

5. the RP validates the token and logs the user in

The service provider is a dummy express.js app that uses passport.js and an OIDC strategy as its auth. middleware to talk to the IdP. To authenticate the user, the IdP serves a static page that allow the user to select in a graphical and user-friendly way the authentication context and its secret identity (index in context + key). Then by clicking on the "Go" button, the selected input is forwarded via websocket to a local DAGA daemon[41] whose role is to conduct the proof of knowledge with the DAGA cothority and to assemble the authentication message.[42] Once done the message is fed into the websocket channel back to the browser that will upon reception POST it back to the IdP. The IdP will proxy the request to the DAGA cothority and receive the final linkage tag in return.

[41] As we can see in the picture, at no point the private key leave the system of the user.

[42] This mechanism was introduced mostly for convenience, to avoid having to port the entire DAGA library and client to JS code.

There were other competing alternatives that are worth mentioning and that were kept in case of trouble implementing the OIDC solution.

- design from scratch our IdP as a kind of certificate authority issuing short lived JWT or x.509 certificates for the users. and have the users authenticate to the 3rd-party service through HTTP header (JWT) or TLS client-side authentication (x.509).[43] To build such CA we even considered using other cothority projects such as ftcosi to make it fully decentralized.

[43] we can note too that there exists readily available passport.js strategies to authenticate users via these certificates.

- finally in last resort fallback to a custom protocol and offer a passport.js module to interface with it. (seems as difficult to learn and implement, if not more (need to modify cothority too), than the other options and restrict the set of users to node.js apps)

4 Results

The DAGA cothority was simulated both locally and on DETERLab facilities using the simulation tools provided by Onet.[8][1] The local simulations were run on a 64-bit x86 machine running Debian 9[2] The DETERLab simulations were run with *pc2133* nodes[3] running Ubuntu 14.04 and connected to the same LAN, where a delay of 100ms was added between each nodes.

[1] DEDIS. Cothority network library., . URL `https://github.com/dedis/onet`

[2] CPU: 8 * i7-4710HQ @ 2.50GHz RAM: 16GiB

[3] CPU: 4 * Xeon-X3210 @ 2.13GHz RAM: 4GiB

As was done in the previous works[14, 19] we varied the number of clients from 2 to 32768 by power of 2 and repeated the experiment for cothorities of size 2,4,8,16 and 32.

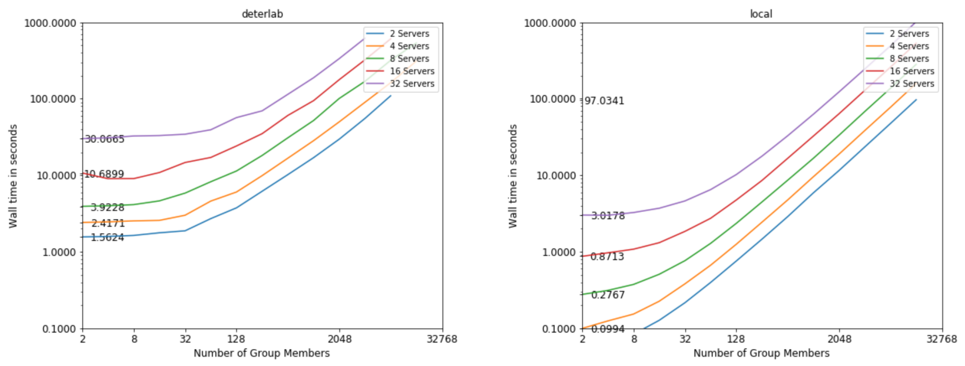
As a key difference to the simulations performed by [14] and [19] the simulations that were performed here are featuring real network communications between different (remote) processes.[4]

[4] even for the local simulations, all conodes are living in their onw process and all communications are going through the network stack

4.0.1 Authentication time

Fig. 4.1 shows the total wall time needed to authenticate with the cothority.[5]

[5] PKClient included, i.e. from the start of the client protocol until the final tag is received.



(a) deterlab simulation

(b) local simulation

Figure 4.1: Authentication wall time

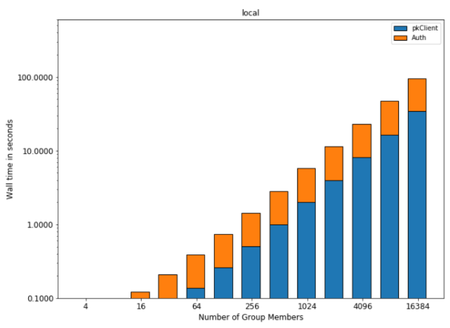
We can clearly observe the impact of the 100ms delay existing in the DETERLab setting. Comparing the results of the local simulation with the one of [19] we can observe a small overhead, probably because now we are running a real cothority (overhead of the cothority framework and network communication). Hence in all logic when comparing it to the one of [14] we notice improvements of the same order as the one noticed in [19].[6] We can see that even when running our simulation in a real world setting (DETERLab) with network delays, we obtain authentication times comparable to or only slightly slower than the ones obtained by [14]

[6] (10X) for the same reasons, notably moore's law

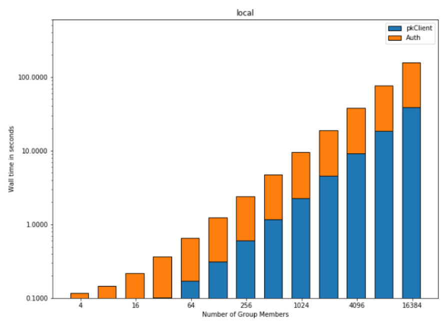
PKClient, Auth, and Total time

Fig. 4.2 decompose the total authentication wall time[7] into the time needed to build the proof (blue) and the time needed to process the request (orange).

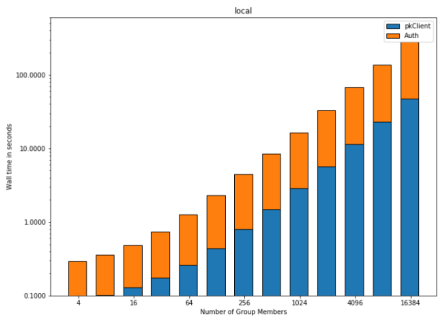
[7] Only local simulation shown, in deterlab similar with only difference being that at beginning the 100ms delay dominate before becoming insignificant later, same difference as in Fig. 4.1 above.



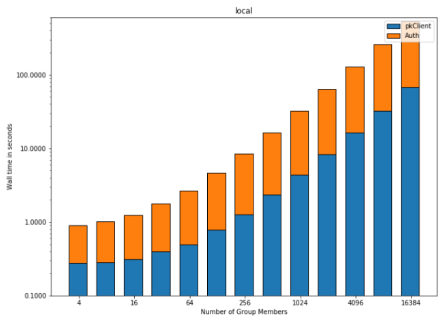
(a) 2 servers



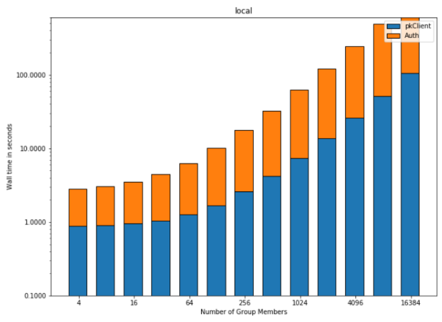
(b) 4 servers



(c) 8 servers



(d) 16 servers



(e) 32 servers

Figure 4.2: Authentication time decomposed

4.0.2 Traffic

Only the total[8] Server traffic was plotted.

[8] the client-server or server-client traffic should not change much in comparison with [19].

Fig. 4.3 shows the sum of the data sent by each server during the Authentication of a user.

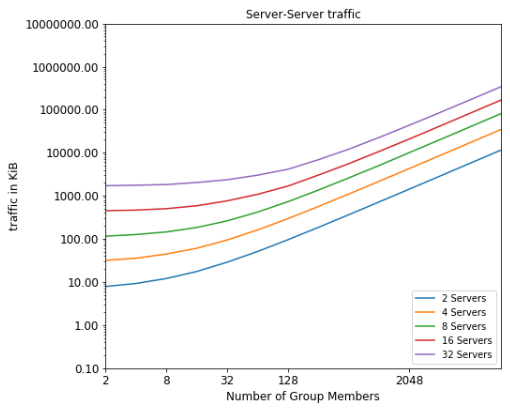


Figure 4.3: Total Server traffic

Finally we share the conclusions that " all forms of DAGA traffic grow linearly in the number of clients and nearly linearly in the number of servers "[14] hence our results are coherent.

5 Conclusion

This project demonstrates the feasibility of the democratization of DAGA as an anonymous authentication mechanism by both

- allowing every system able to speak OIDC to make the move (pretty much every system) without pain
- and showing that it is possible to offer a user-friendly interface

This was done without introducing alien ideas into the DAGA library and Cothority making them easier to maintain and reuse for other purposes. Nothing prevent us to tie DAGA more closely to the POP framework, everything is already usable, it lacks only the little bit of boilerplate extracting the keys from a party transcript.

Still it is far from being perfect and finished, notably from a "features" point of view, we haven't yet defined ways to allow 3rd-party services to evolve the authentication contexts (add / remove users, revoke context, deny authentication if user exhausted some quota of authentication etc.). From a security point of view we can list the following things that would need our future attention:

- currently, the authentication messages can be replayed...
- we would need ways such as memguard[1] to protect the storage of secrets and keys in memory.
- currently the state of the service contains sensitive information (per-round secrets, secret-keys of the daga servers associated with each contexts, etc.)that are persisted into bbolt... We propose to make it fully stateless if possible by deriving every needed secret from the master secret key of the node and other information and "storing" more state in the clients.
- As already said the server-side code of the library would need some reviews too
- We noticed that the deniability property as it is currently defined in the paper doesn't totally hold. At the end of a successful authentication every servers obtain the list of all the NIZK proofs that state they all did their work correctly. Since those proofs are verifiable by anyone and we are in an anytrust setting then for sure someone did authenticate (if not that would mean that the honest server did not follow the protocol this could qualify as a

[1] https://github.com/awnumar/memguard

dishonest behavior..). hence we would need to modify the implementation to periodically add some noise, release proof transcripts that are nothing but simulations etc..

Bibliography

[1] Announcing dex, an Open Source OpenID Connect Identity Provider from CoreOS | CoreOS. URL `https://coreos.com/blog/announcing-dex.html`.

[2] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, volume 3958, pages 207–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-33851-2 978-3-540-33852-9. doi: 10.1007/11745853_14. URL `http://link.springer.com/10.1007/11745853_14`.

[3] Dan Boneh. The Decision Diffie-Hellman problem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, Lecture Notes in Computer Science, pages 48–63. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-69113-6.

[4] Maria Borge, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 23–26, Paris, April 2017. IEEE. ISBN 978-1-5386-2244-5. doi: 10.1109/EuroSPW.2017.46. URL `http://ieeexplore.ieee.org/document/7966966/`.

[5] Jan Camenisch and Markus Stadler. Proof Systems for General Statements about Discrete Logarithms. Technical report, 1997.

[6] CoreOS. OpenID Connect. URL `https://openid.net/connect/`.

[7] DEDIS. Advanced Crypto Library for Go, . URL `https://github.com/dedis/kyber`.

[8] DEDIS. Cothority network library., . URL `https://github.com/dedis/onet`.

[9] DEDIS. Scalable collective authority., . URL `https://github.com/dedis/cothority`.

[10] Andrew Lindell. Anonymous Authentication. *Journal of Privacy and Confidentiality*, 2(2):24, 2011. doi: 10.29012/jpc.v2i2.590.

[11] Rob Pike, Ken Thompson, and Robert Griesemer. The Go Programming Language. URL `https://golang.org/`.

[12] Stuart Schechter, Todd Parnell, and Alexander Hartemink. Anonymous Authentication of Membership in Dynamic Groups. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Matthew Franklin, editors, *Financial Cryptography*, volume 1648, pages 184–195. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-66362-1 978-3-540-48390-8. doi: 10.1007/3-540-48390-X_14. URL `http://link.springer.com/10.1007/3-540-48390-X_14`.

[13] Latanya Sweeney. k-ANONYMITY: A MODEL FOR PROTECTING PRIVACY. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, October 2002. ISSN 0218-4885, 1793-6411. doi: 10.1142/S0218488502001648. URL `http://www.worldscientific.com/doi/abs/10.1142/S0218488502001648`.

[14] Ewa Syta. *Identity Management through Privacy-Preserving Authentication*. PhD thesis, Yale, December 2015.

[15] Ewa Syta, Benjamin Peterson, David Isaac Wolinsky, Michael Fischer, and Bryan Ford. Deniable Anonymous Group Authentication. Technical Report 1486, Department of Computer Science, Yale University, February 2014.

[16] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society. ISBN 978-1-891562-46-4. doi: 10.14722/ndss.2017.23171. URL `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/measuring-small-subgroup-attacks-against-diffie-hellman/`.

[17] vibro. OAuth 2.0 and Sign-In, January 2013. URL `http://www.cloudidentity.com/blog/2013/01/02/oauth-2-0-and-sign-in-4/`.

[18] Arthur Villard. Deniable Anonymous Group Authentication, 2017. URL `https://github.com/dedis/student_17_pop_fs`.

[19] Arthur Villard. report_pfs_pop.pdf. Technical report, 2017. URL `https://github.com/dedis/student_17/tree/master/pfs_pop`.

[20] Samuel Warren and Louis Brandeis. The Right to Privacy. *Harvard Law Review*, 4(5):193–220, December 1890. doi: 10.2307/1321160. URL `https://www.jstor.org/stable/1321160`.

[21] David Isaac Wolinsy. DAGA as a service., 2014. URL `https://github.com/davidiw/dagas`.

6 *Annex*

Instructions to run the proof of concept.

Prerequisites:

- a working Go installation[11][1]
- a working docker and docker-compose installation
- retrieve github.com/dedis/student_18_daga and install the daga-client and dagadaemon cli
- add the following two aliases to /etc/hosts:

  172.18.0.1 opapp.poc

  172.18.0.1 rpapp.poc

  where 172.18.0.1 should be the IP of your host computer as seen from the future docker containers, adapt if it is not the case.

then:

1. cd in the student_18_daga/PoC directory
2. *make*
3. follow the instructions on screen

[1] Rob Pike, Ken Thompson, and Robert Griesemer. The Go Programming Language. URL `https://golang.org/`