Implementation of a Petnames System in an existing chat application

Jessica Tallon, Christine Lemmer-Webber, Randy Farmer

Table of Contents

A petname system allows for mapping personally meaningful names onto otherwise non-meaningful decentralized identifiers. In this paper we present our experiences from implementing a petname system for an existing chat application. We introduce the necessary mechanisms to retroactively introduce a petname system, the solutions provided, and future expected work.

1. Introduction

Goblin Chat is a peer-to-peer, capability secure chat system built on top of Spritely Goblins. While compactly built and a successful demonstration of object capability architecture, its initial naming system was undesirable. The initial naming system allowed users to set their own name with no checks to avoid name collision or verify whether someone else also went by that name in the chatroom. This lead us to search for a better naming system which would address some of these shortcomings but would still retain the peer-to-peer, decentralized nature of Goblin Chat.

1.1. Motivating examples

Alisha, Ben, Carol, and Mallet are users of Goblin Chat. Alisha wants to speak to her friend Ben.

Ben would also like for Alisha and Carol to become friends. Mallet is a malicious entity who would like to interfere with Alisha's social life, perhaps by masquerading as another user or by phishing Alisha into giving away details about her private life, or perhaps tricking Alisha into giving Mallet money when she meant to give Ben or Carol money.

Initial versions of Goblin Chat successfully demonstrated several valuable properties: Goblin Chat has, from the beginning, been a fully peer-to-peer and decentralized chat system. From the beginning of Goblin Chat, messages would be safely authenticated as coming from a particular participant on a programmatic "object identity" level. For example, say Ben's chatroom identity is represented under the hood as:

```
ocapn:s.onion.wy46gxdweyqn5m7ntzwlxinhdia2jjanlsh37gxklwhfec7yxqr4k3qd/
xz4ey8z5q4ag99zvlnaqyjfy50y2gaxcpi5w2rc68plrekskjviu0mb0vb5hkzed
```

There is no way in Goblin Chat for Mallet to produce messages claiming to be from the above address. In this sense, Goblin Chat is very robust in its authentication safety.

However, this kind of safety is near useless to human beings, as the user interface representation of a user's name in early versions of Goblin Chat is whatever that user chose to call themselves, their "self proclaimed name". While Goblin Chat would be able to distinguish between a message coming from Ben's chatroom identity and Mallet's chatroom identity, nothing would prevent Mallet from choosing the same name as Ben, leading to two "Ben" users in channel. In this situation, Alisha has no clear path to distinguishing between messages coming from the "Ben" she knows and the "Ben" provided by Mallet.

Could we do better in building a decentralized naming system? Zooko's Triangle is a trilemma which says that a name can have no more than two of three of the following properties:

- **Human meaningful:** A name which is meaningful to the specific user
- **Globally unique / Secure:** A unique name which avoids name confusion
- **Decentralized:** Not reliant on a central authority.

Dropping **decentralized** would not be an option for Goblin Chat, since building a system which is as decentralized as possible is a fundamental goal. Early Goblin Chat had, in a sense, two different kinds of names: **decentralized + unique** names like the `ocapn:...` identifier shown for Ben's chatroom identity earlier, but these are not human meaningful. Meanwhile, the *self proclaimed names* of "Alisha" and "Ben" are **decentralized + human meaningful**, but they are not *globally unique / secure*; Mallet is able to take advantage of this to masquerade as Ben.

If no name can achieve all three properties at once, how can we do better?

1.2. Petname systems

When building a decentralized system which have identifiers for people being secure and distributed (e.g. URIs of tor onion nodes), then there is the problem that they stop being human meaningful. This problem can be solved by introducing a petname system.

The petname system tries to address this by introducing different types of names that when combined provide a ways of being meaningful and unambiguous, these types are:

- **Petnames**: Local names which are individual to each user for their own use.
- **Self-proclaimed names**: Public name which the user gives themselves to be referred to them.
- **Edge names**: Shared names which users give other users and then can share with others to

help discovery and corroborate who a user is.

Each one of these have different strengths and weaknesses.

Petnames are kept locally and given by the user for other people, because of this the names are meaningful on the individual level for that user. A user could for example name a person "dad" which is meaningful to that user, but obviously not for others.

The downside of petnames are obviously that a new user would be nameless until a user has assigned a petname to them. This is where self-proclaimed names come in, each user gives themself a name which they want to be publicly visible. This name would not be tailored to each individual user in the same way we saw the example of "dad" above, however it will hopefully give a useful name until a user assigns their own petname.

These names are however not without problem, self-proclaimed names suffer from not being unique. In a distributed system with no central naming authority, multiple people could provide the same self-proclaimed name. There could be a chatroom for example, with multiple users called "Ben". This could even be done intentionally by a malicious user who is trying to impersonate someone else.

When users are presented in a UI, a display mechanism is required to help users disambiguate names that are identical (or even similar).

Finally, edge names. These names are proposed names given to users with the intention of sharing them with others. These could be in two forms:

- By individual users who share them with friends, family, etc.
- As a directory with a collection of users (e.g. a directory of colleagues at work).

These edge names can be used to help further identify users by supplying context, as well as corroborate who users are when there is no central authority asserting the veracity of names.

It's worth mentioning that while this paper focuses on implementing a petname system for users of a chat program, it can be applied to other context's too. The petname system could for example be used for naming the chatrooms where the creator of the room could give the chatroom a self-proclaimed name and each user could give it a petname.

2. Implementation

2.1. Goblin Chat

The chat application that is being used to implement this is called Goblin Chat which is built on top of Spritely Goblins. Goblin Chat is a GUI chat application which allows users to speak to each other in chatrooms. Each chatroom can have any number of users participate and for users to participate in any number of chatrooms.

2.2. The Spritely Goblins Framework

See the Spritely Goblins paper for a detailed description of the underlying framework, but now provide a brief summary to help understand how we implemented the petnames system.

Goblins is a framework to easily build peer-to-peer distributed applications. This is achieved by building on top of an actor-model which has objects (actors) communicate with each other by passing messages. These messages can contain responses which are communicated back to the sender by use of promises which are resolved with a value. Objects can fulfill multiple functions by

means of provided multiple "methods".

A computer may manage multiple objects which can communicate with each other, including across networks. Goblins itself is network agnostic however, Goblin Chat currently uses the tor onion network to communicate: underlying each object is an "onion address". The details aren't especially important to this paper, however it's important to mention that each address is unique and they persist across sessions.

Goblins also takes a capability security approach to authorization. A Goblins object by default has no access to other objects. Access (the ability to send messages to an object) comes in the form of a capability grant: essentially a (remote) object reference and procedure call bundled together. These capabilities are revocable, and provide limited access to the object.

2.3. Building the underlying petname system

2.3.1. Petnames

If a user has assigned a petname for a given contact the UI will display the petname for the contact instead of any available names such as a self-proclaimed name.

Goblin Chat also allows users to reference other users when writing a message by entering the petname in the text entry box. But when sending this over the network we want to use the onion address for that user to other parties in the chatroom, so that each of them can translate it to an appropriate meaningful name.

We implemented this with a bi-directional hashmap that can be serialized to disk and restored. This provides a fast lookup that that will persist across sessions using existing Goblins and language features. This has provided a simple, yet effective way of providing all the functionality we need to represent petnames. We implement this with a bi-directional hashmap that can be serialized to disk and restored.

2.3.2. Self-proclaimed name

In Goblin Chat each user is implemented as an actor with several methods, one of which is a getter to retrieve the user's current self proposed name.

This name is cached by each client once retrieved to reduce round trips and network overhead, however this could be a problem if it's implemented so that a user can change their self-proclaimed name. This particular problem is not addressed in this implementation.

2.3.3. Edge names

Goblin Chat allows users to assign edge names to users and then share these to specific users of their choosing. When a user begins to share their edge names with another user this can be thought of as a subscription to the supplier's edge name database. Any additions, modifications or deletions to the database are synchronized.

We found that the edge name implementation is by far the most complicated aspect of this to implement and with the most number of decisions to make. Those are decisions are discussed later in the paper, however our implementation features:

- One set of edge names which can be shared with many users (i.e. the user can only give one edge name to a user)
- Edge names are offered to a user by the user who is sharing them.

- The database is synchronized when both users are online and the database kept up to date
- Users are not assumed to always be online.
- A user can revoke a subscription when they want.

The edge name database is implemented much like the petname database in that it's a hashmap from a user's onion address to an edge name. This is serializable to disk and restorable.

1. Initiating a subscription

   When a user wants to share their edge names, a specific actor is generated for the communication between the sharing user and the recipient. This actor is given to the recipient (their user object) who can then send a message with their own generated actor (this actor is used by the sharer to send updates to). These actors can then work together to keep the edge name database in sync.

   In a system which does not use capability security, one would have to take care that only the sharer could send updates and that they were sharing it with the intended recipient. Without capabilities, edge name databases could exploited to either supply false data to the recipient, or eavesdrop on the contents of the edge name database.

2. Synchronization

   As elided to in initiating the subscription, the sharer actor and the recipient actors which were created, are used to keep the database in sync. In our system it's assumed users go online and offline and so the synchronization of the edge name database needs to take this into account.

   If both users are online at the same time, when a change is made to the database it looks through all the subscribers and sends an update message to their recipient object with the new edge name database.

   When a user is offline, they are not sent any messages and if the user comes online it will send an "I am available" message to the sharer object know it should send any update if it exists.

   This version works by sending a copy of the entire database over rather than just a changeset. A better implementation of the update mechanism would use a crdt or equivalent, however this wasn't readily available when implementing this implementation and building one beyond the scope for this project.

3. Ending a subscription

   Due to privacy reasons, we are taking the step not to explicitly notify the recipient that they are no longer sharing edge names. We simply destroy the sharing actor and it will appear to the edge name handling code on the recipient side as if they are offline.

   If the sharing user wishes to share their edge names again, they can follow the initiating procedure outlined above again and a new subscription will be created.

2.4. Building the GUI

The petname system requires that the user of the application is able to distinguish between the different name types. If they are not able to easily tell the different name types apart, then it leaves the user susceptible to attack where a malicious user could try to pass themselves off as a contact the user has a petname for.

To do this we have built a basic GUI where each name has distinguishing features which make it

clear which type of name is which. We also only display one name for each user. If we have a petname, that is the name we display, otherwise we display the self-proclaimed name for that user. Edge names can be viewed in Goblin Chat by looking at the users profile (more on that later).

2.4.1. Petnames

The petname is displayed with a solid box with the petname the user choose inside.

2.4.2. self-proclaimed names

The self-proclaimed name has a box with a dashed line surrounding it. The self-proclaimed name is inside, however it's also proceeded by a question mark to make it clear that this is a self-proclaimed name.

2.4.3. Edge names

The display of edge names is kept to the profile page, this is also where a petname and an edge name would be set for the user. The edge names themselves need to convey more information than the other two names, this is:

- That it is indeed an edge name
- Who the edge name is from (being careful to identify the sharing user with the petname or self-proclaimed name).
- The edge name the sharing user is proposing.

Each name follows the same pattern of:

[Icon] [Name of Sharer] -> [Edge name]

The icon in our system is a common "sharing" icon, three dots connected with two lines. We hope this will illustrate that this name has been shared with the user. It also includes the name of the sharer, with an arrow graphic pointing to the edge name the user is sharing.

This name is wrapped in the same style box as the name of the sharer

This can look one of two ways:

1. Shared by someone with a petname

2. Shared by someone with a only a self-proclaimed name

2.4.4. Profile page

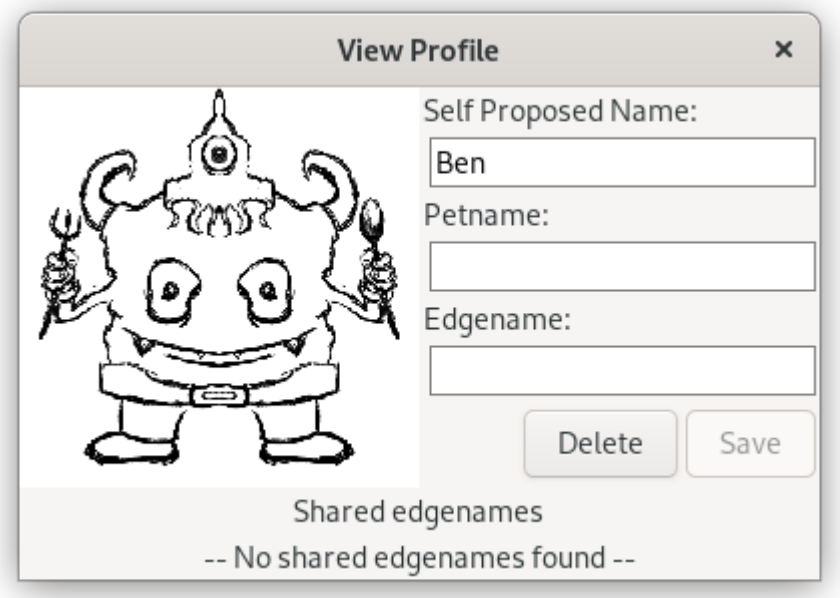
As mentioned above, the profile page is where a user can go to set a petname or an edge name for a user. It's also where sharing or revoking their edge name database can be done as well as seeing all the edge names for that user.

Since self-proclaimed names (and even edge names) are not unique, it brings in the question of disambiguation. If two users in a chatroom are called "Ben", the user needs a mechanism to tell these apart. The way we're addressing that is to display a graphical monster which is generated from a salted hash of their onion address.

Part of the reason for selecting a monster figure as shown in the images below, is because they don't look like people, but fictional characters. We didn't want people to feel like the monsters should be representing how they look, so we designed them to look fantastical and non-human.

This monster graphic will always be consistent for a given user and has the attribute that due to the salt, no other users will be able to predict what the monster picture is. This is important as there are limited monster combinations and with enough time an attacker could generate many onion addresses until one generates the monster they wish. This could be used to impersonate someone. The salt means that each monster will be different for different people and that it cannot be deduced which monster the user has as the salt is kept private and local to each user.

Here are two images of two profile pages for two different users with the same self proclaimed name:



2.5. Results, Surprises and other Lessons Learned

There were many open questions which were discovered while implementing the petname system in Goblin Chat. These often didn't have a clear answer for which is best. Something that is lacking from our work so far is clear user testing, the need to distinguish between the different name types and understanding the benefits and drawbacks is important to get the most out of the system and stay safe. User testing is therefore important and as the petname system is not widespread (yet), some explanation or introduction to users would probably help.

2.5.1. The addition of edge names

Edge names are the most complex addition. They require a subscription and syndication mechanism. They also are the most unfamiliar part of the system to users requiring careful UI design and explanation to how they operate and the purpose they are supposed to serve.

Usually users who select a self-proclaimed name, select a sensible one which should be meaningful to most people who interact with them. With other factors which can be used in conjunction to disambiguate users (discussed below), it should also be possible to help against *some* cases of name confusion (where multiple people have the same name).

Edge names are also an avenue for abuse, as explored below.

This area of the petname system would benefit from further exploration and investigation, especially with the focus on how new users to the system use them and how useful or not they might be.

1. Directory style edge names

   There is, of course, both the directory style edge names where one could subscribe to directory of edge names such as an employer or municipality might provide vs. individual users sharing their own databases.

   With the directory style edge names it could be implemented in a much simpler way as one could assume it'd always be online and could be queried on demand instead of keeping a local copy. This would simplify the implementation greatly.

2. Sharing a single contact

Another solution to a user wanting to share another user with someone else is to implement contact sharing, similar to what one might find on their phone. This would allow a one time sharing of a user entry from one user to another without synchronization and might address some of the uses of user to user edge names.

2.5.2. Subscribing to edge name databases

The implementation that Goblin Chat took was to provide a way to users to share edge names with specific users of their choice by selecting a user and selecting the "sharing edge names" option in the menu, however there are many mechanisms for sharing the edge name database that could be chosen. Below are some of them which we had considered when implementing edge names.

1. Sharing a capability

   With a chat application or similar, it's possible to share a "capability" in a chatroom which allows a user who has it to subscribe to the database without further interaction from the user sharing it. This capability could also be revoked by the user to make it no longer functional if they wish to stop new people using it to subscribe.

   The UI for this could be displayed as a user clickable widget which starts the subscription. When the user interacts with the widget in an affirmative manor, a request to subscribe to the edge names is sent to the person, this could be automatically accepted on the user's side.

   Goblins itself provides enough tooling to do this easily, by sharing a new object which has this capability with a method on it to subscribe. In a traditional system, this could however be done by creating a key pair, the public key would then be the capability which would be given out and those who get it could sign a "subscribe" request to the user, the user could then check the signature and know it's valid and begin sharing the edge names with the requester. If the person who gave the capability in the future wants to revoke the capability they could simply stop honoring signatures from that key.

2. Sending a request to start sharing

   This is the method we went with due to the simplicity of implementing it.

   This involves the user selecting which users they wish to share their edge names with. There is a question for what happens when they have chosen to share their edge names with a specific user, they could have some form of confirmation to accept the sharing, or it could simply begin sharing the edge names with the user automatically. If a confirmation dialogue was chosen, it could be abused by a malicious user continuously spamming sharing requests.

3. Requesting access to edge names

   This is similar to the above, however it's the person who'd like to receive the edge names who would request them of someone else. This could work in tandem with the above method, or on it's own. With this method a request which the user who would be sharing their edge names would have to accept in order to share begin sharing them.

4. Sharing edge names automatically to "friends" / "following"

   This method would be build around a relationship which would be established within the application. If this were a micro-blogging service, this could be the users that the user is "following" or in a chat app the user might be able to "friend" other users.

   This brings up questions making sure the user is aware this action also initiates sharing of their edge name database, as well as what other action it performs. If the relationship it's based on is otherwise asymmetric, e.g. "following" a user, would the user on the other end

have any notification and possibility to accept / deny having the edge names shared with them.

5. How to handle unsharing/unsubscribing to edge names

   There is probably a time when a user would like to stop sharing their edge names with a specific users. When this happens there are multiple things which should be considered when implementing this functionality, such as privacy and harm from stale data.

   1. Notification upon stopping sharing them

      This would involve when a user stops sharing the edge names, it notifies the person and they can handle it, marking the data as stale, removing the data, etc.

      This however has privacy implications as the user may not intend to notify the other user that they're stopping sharing them (similar to an unfollow request). This could be solved by the notification simply not telling the user, but performing the correct actions behind the scenes. This relies on the client working in a specific way and that cannot be trusted to be the case.

   2. Sending a final update which empties the database

      This would not send a specific notification or information that the user is no longer sharing edge names with them, however it would ensure the data has been removed from the user, meaning no stale data should persist. This however brings issues of the above that with heuristics it could be possibly to reasonably determine that the user has stopped sharing edge names with them.

   3. Do nothing

      This protects privacy at the cost of stale data. It gives the recipient user no indication that the sharing user has stopped sharing the edge names, they simply stop updating them.

      The risks of stale data can be harmful. For example, if a person knew a transgender user before they came out, they probably would have shared the name they knew them as before transition. This could cause real harm to everyone involved by not being removed or updated.

      The way we solved this was to not update the user, but to mark all data as "stale" if there has been no update to it after 30 days. This stale data does not appear in queries for edge names. This would also ensure those who've lost access to their account or stopped using it would not have stale data they have shared surfaced.

3. Conclusion

Adding edge names to Goblin Chat with petnames demonstrates the complexity of implementing user-to-user sharing of edge names in contrast to the relative each of implementing just self-proclaimed names and petnames.

The edge name system currently implemented does not provide directory style edge name subscriptions. These could be implemented with an active query feature assuming the publisher would be always be online. This would greatly simplify the implementation and identified issues, such as surrounding stale data and how subscription initiation flow.

The full-featured petnames system requires significant additional user interface testing, especially focused on name display patterns and the additional complexity/benefit trade-off for edge names as described in this paper.

This paper doesn't cover new user on-boarding in a world where the petname system is not a common feature of applications. This is something which requires further research and care.

Despite certain unanswered questions around edge names, we have found that implementing a petname system is otherwise a fairly straightforward and pleasant process. Additional best practices for petname systems will come with additional implementations, which we recommend for other decentralized networking technology projects.

4. Thanks

The work implementing the petnames system in Goblin Chat and writing this paper has required a lot of work, which could not have been done without the support and funding from the NLNet Foundation.