xTalk: A Nomicle-Powered, Inter-App Messaging System

1 JUN 2022

Ali Mahouk
Email: ali.mahouk@pwc.com
xTalk: mail@alimahouk

The Initiative for Novel Research & Applications in Computing

1. Synopsis

Sending a message across a network is relatively easy when the recipient's IP address is known to the sender or when a central server is present at a static IP address to authenticate new and existing clients, as well as handle message routing and storage. It becomes especially challenging to achieve the same objective in a decentralised network where the sender does not know the IP address of the recipient, but does know some other mnemonic information that might uniquely identify them. This paper proposes a system that facilitates message exchange across a decentralised network by allowing messages to be addressed to arbitrary, mnemonic tokens rather than IP addresses.

2. How It Works

xTalk is a decentralised messaging service intended for use by applications. It is the first system based on the new Nomicle platform, which is a decentralised identity system. [1] Messages are encrypted and short (payload is limited to 140 bytes) as xTalk is not intended for use as a communication layer, but rather as a bootstrap layer, i.e. peer-to-peer apps use xTalk to find each other and then set up their own dedicated connections to talk over their own protocols. However, nothing prevents apps from using xTalk for communication as well provided they kept their messages within the aforementioned payload limit.

xTalk runs as a daemon in the background on a computer (a prerequisite is having the Nomicle program suite installed as well); apps on the system can connect to it over TCP to send and receive messages to and from other apps on remote hosts as well as the local system. It uses a simple textual protocol along with a very flexible and robust recipient addressing scheme. xTalk then handles encryption and delivery as well as sending back a receipt once the message is delivered to the recipient. It is designed to allow apps to 'fire and forget' without having to worry about the intricacies, complexities, and potential hostilities on a decentralised network.

2.1.Using xTalk

The system can be configured to listen on a particular TCP port for apps that require its services (depending on the use case, it may be configured to only accept incoming connections from the loopback interface and/or from the network). The protocol between apps and xTalk is text-based. Apps that wish to receive messages register with the daemon upon starting up by providing it with a list of service

identifiers that tells the daemon what kinds of messages the apps are interested in. Service identifiers are arbitrary strings but there ought to be some sort of convention that all developers would adhere to, e.g. 'mail' for applications that handle email or 'http' for applications that handle HTTP requests. An application may provide the identity token of the recipient it wishes to reach along with a service identifier (i.e. which application to deliver the message to on the recipient's machine), and an arbitrary string, which is the message that the app wants to send. xTalk will use the Nomicle private key of the current machine along with the public key from the nomicle of the requested recipient to encrypt the passed string, package it into a message block, and sign it. The message is then broadcast to other nodes. Each node can check whether it is the recipient of the message, and if so, it decrypts the message, verifies the signature, and delivers it to the app on the recipient's machine. Signed delivery receipts are sent back to the sender, which serve to help clean up the network of copies of the message as it has already reached its intended recipient. If the nomicle of the recipient is not found in the local Nomicle repository, the message cannot be sent. It is saved to the disk and the identity of the recipient is added to the Nomicle probes file. The system periodically checks pending messages to see if their recipients' nomicles have been found, in which case that message gets sent off.

A message must be short. The limit is 140 bytes so as not to take up too much bandwidth and disk space. Typical message contents could be the IP address(es) and port number(s), along with a TTL (time to live), of the sender so that the recipient may connect directly to them and they may then communicate using whatever protocol they require over their own socket connection. Despite message payloads being encrypted, the xTalk messaging function is not intended to be used directly for the transmission of security-sensitive information owing to the fact that you do not know who may own the recipient identity at the time that you are about to send anything. Hence, it would be wise for apps to carry out their own checks of who is on the receiving end and handle such sensitive transmissions themselves.

xTalk is not designed with the goal of anonymity as a message includes the identifier of the sender (encrypted within the payload so only the recipient can see it). However, it is possible to create a fork of the project that deals with messages anonymously.

2.1.1.Read Receipts

When a message reaches its intended recipient, they can publish a read receipt. This lets nodes on the network know that they can delete their copy of that message to stop it from propagating any further. Receipts include a hash of the message to which they refer and are signed by the private key of the recipient, so other nodes can verify the message in question and use the public key in the recipient's nomicle to verify the signature in the receipt. Copies of a receipt may remain on the network for up to thirty days. Any receipts that are older than thirty days are ignored by nodes and not propagated further (copies in the local repositories are periodically purged as well).

2.1.2.Recipient Addressing and Service Identifiers

xTalk addresses should generally adhere to the format 'service@user'. They do not map to IP addresses and are not tied to hardware, i.e. xTalk makes no assumptions about anything underneath the application layer; the task of deciding how to send the messages (e.g. using IP over the internet) is left to the implementation of the xTalk client. Multiple devices may be set up to receive messages sent to a particular address. There is no guarantee, however, that a message will always be received by all those devices since the first device to receive it will publish a receipt and that might prevent copies of the message from continuing to propagate until they reach the other devices.

The part before the '@' symbol is the service identifier (SID) and allows applications to indicate what their message is all about in order for apps on the receiving end to know whether they should attempt to parse it or not. SIDs are arbitrary, case-insensitive strings and entirely convention-based. The only obvious restriction is that they must not contain spaces, '@', or ',' (comma) characters anywhere within them. Anyone can come up with their own SID to use in their own apps. Examples of generic SIDs might be 'mail', 'http', 'ftp', 'telnet', etc. Since multiple apps might be simultaneously interested in the same SID(s), the same incoming message will be delivered to all those app instances.

The part after the '@' symbol is the user's Nomicle identifier.

2.1.3.Sending a Message

The first step is to make sure the Messenger is running. From your application, you create a new TCP socket and connect to 127.0.0.1 at port 1993. Sending a message requires three elements: a recipient, a

service identifier, and a message string. The recipient and/or SID may be omitted in some scenarios (See §2.1.5 Inter-App Communications). You write the following to your socket:

```
to: someservice@bob\r\n
body: Hello, bob!\r\n
\r\n
```

To send a message without specifying a service, do it like so:

```
to: @bob\r\n
body: This is a generic message.\r\n
\r\n
```

Each field of a message is delimited by a carriage return followed immediately by a line feed. The end of the message is indicated by a double instance of \r\n. Once you have written the above text to your socket, the Messenger will reply with the new identifier generated for that message, which your app may retain:

```
ref:
81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce
9\r\n
```

You may use this reference ID for either of two things:

* Following up on the status of a message to know whether it has been delivered or not.
* Creating message threads by replying to a specific message.

To follow up on whether a message you've sent has been sent or delivered yet, you send the following message:

```
status:
81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce
9\r\n
\r\n
```

The string passed in is the identifier of the message (you would have needed to retain it after the Messenger returned it when you sent the original message) whose status you are inquiring about. The Messenger responds with one of the following integer values:

* 1: The message is still pending. This means the Messenger is still waiting for the nomicle of the recipient to show up in the Nomicle repository.
* 2: the message was successfully sent but has yet to reach its recipient.
* 3: the message was received by the recipient.

```
status: 2\r\n
\r\n
```

Your application may choose to send a message in reply to a specific message. This extra field is simply a convenience for apps and bears no semantic meaning to the Messenger (i.e. the Messenger does no verification to check if the message being replied to actually exists).

You may reply to a message like so:

```
to: telnet@bob\r\n
re:
81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce
9\r\n
body: Thank you for the birthday wishes.\r\n
\r\n
```

The string value in the 're' field is the identifier of the message you are replying to.

2.1.4.Receiving a Message

To receive messages, your app needs to register what services it is interested in handling with the Messenger. You register by passing in one or more SIDs as a comma-separated list:

```
interest: telnet, http\r\n
\r\n
```

To receive messages sent without an SID specified (to: @someone), you use the special '*' SID. This is the wildcard SID used for generic messages. Do not confuse this with meaning 'I want to handle any and all inbound messages, regardless of their SID'.

```
interest: *\r\n
\r\n
```

You may also include '*' as part of a list of other SIDs:

```
interest: telnet, *, http\r\n
\r\n
```

Every time you send an 'interest' message, the SIDs you provide will overwrite any previous interests you registered.

The Messenger will reply with '0' if the registration was successful or '-1' if an error occurred:

```
interest: 0\r\n
\r\n
```

When a message arrives for one of your registered SIDs, your app will receive it in format similar to the following:

```
from: ali\r\n
ref:
81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce
9\r\n
time: 2020-02-06 22:52:01\r\n
body: Call me when you receive this.\r\n
\r\n
```

If this message was in reply to one you sent earlier, the 're' field will be included with its identifier:

```
from: ali\r\n
ref:
81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce
9\r\n
re:
8bea037d6e1cadf853d712f075fe52f471eca1a736c5ef5273fd7708e10fb57
1\r\n
time: 2020-02-06 22:52:01\r\n
body: Call me when you receive this.\r\n
\r\n
```

A message is considered delivered as soon as any running app on the machine receives it. If your app is not running and a different app receives a message for a service your app handles, your app will not receive that message the next time it runs. If messages for a service are still undelivered, your app will receive them as soon as it registers its interest for that service. Once a message is delivered, it is deleted from the local message repository and a delivery receipt is sent out to the network to alert the sender.

2.1.5.Inter-App Communications

xTalk messages are platform agnostic. This means apps don’t necessarily have to communicate with their identical counterparts on other machines or even the same machine. You may very well use a web app to send a short message which the recipient might read via Telnet.

It is possible for apps to send messages addressed to the local user, but with different services before the '@' sign. In this case, you send a message that omits the '@' symbol:

```
to: someservice\r\n
```

This effectively allows apps (or even webpages) to communicate with other apps on the same machine, much like using the loopback interface in networking, without having to worry about intricacies like message delivery (e.g. if the recipient app is not currently running), protocols, and port numbers.

2.1.6.Getting the Local Nomicle Identifier

An app may either read the Nomicle identity file for itself or it may request it from the Messenger:

```
me\r\n
\r\n
```

Response:

```
me: bob\r\n
\r\n
```

2.1.7.IP Address Discovery

One expected usage of xTalk will likely be the exchange of IP addresses to allow apps on different machines to connect directly to one another in a peer-to-peer fashion. Public IP address discovery typically requires the use of STUN/TURN. Since xTalk is likely to have already established its own P2P network, a node can ask one of its peers for its own public IP address. An app can request that address from the Messenger, like so:

```
address\r\n
\r\n
```

The Messenger will respond with the machine's public IP address as well as its private address on the LAN, if one exists, separated by commas:

```
address: 10.21.97.88,192.0.1.1\r\n
\r\n
```

Port mapping, however, is left to the app or the network administrator.

2.1.8.Hosts

The Messenger maintains a list of known xTalk installations in the hosts file (/usr/local/var/xtalk/hosts on Unix-like systems or %APPDATA%\xTalk\hosts.txt on Windows). A user may manually modify this file to add or remove IP addresses and port numbers as they wish. The Messenger uses the broadcast address to discover other installations on its LAN.

2.2.Configuration

Configuration files consist of key-value pairs delimited by newlines. A graphical interface may be created to allow the user to configure their installation without having to delve into the

configuration files themselves. The following aspects may eventually be configurable by the user:

* Port numbers: these would be standardised so there needs to be a good reason to use anything other than the defaults. This might be the case in a custom xTalk setup running on an intranet.
* What directory to use as the message repository.
* Black/whitelist identities: this may be done on a token, public key, and/or trust authority basis. The Messenger can check this list and accordingly accept or discard certain messages based on their sender.

3. Nomenclature

xTalk is pronounced 'crosstalk'.

4. Conclusion

The ability to send and receive messages is a fundamental need for decentralised apps. In almost all cases, the basic set of problems and hurdles that need to be overcome are the same. We proposed a generic system and an addressing scheme that allow apps to send and receive short messages not just to and from other instances of themselves, but other kinds of apps as well without having to handle the complexities and intricacies of the communication themselves, all while maintaining decentralisation.

References

[1] A. A. Mahouk, 'Nomicle: A Decentralised, Identity-Based, Cryptography System,' 2020.