

Hotline Networking

Eric Grosse grosse@gmail.com

version 2020-02-18 latest version will be at github.com/n2vi/hotline

This document makes a concrete proposal for the network link between the Puck and the Broker hotline devices owned by one country. We seek a small, easily reviewed network stack with minimal OS support. It should depend on little in the surrounding office environment.

In the CATALINK architecture, there is a small device called the Puck owned by a hotline Principal that holds all secrets for the end-to-end encrypted hotline. It may only be intermittently powered on, so relies for connectivity on the Broker, a well-connected and always-on relay server, also owned by the Principal but with somewhat less security sensitivity. Brokers talk to other principals' Brokers using a highly resilient network called ROCCS, which might include a mesh RF network and satellites and the Internet. The goal of this document is to describe how Puck talks to Broker.

Puck is expected to run a microkernel such as seL4 plus a small amount of userland software that should be reviewable by people from many backgrounds. A typical TCP/IP network stack would be larger than the entire rest of the microkernel so the CATALINK team identified early on that a simpler networking alternative is a crucial element of technical credibility for our project.

Broker is expected to run a larger operating system such as Windows or Linux or OpenBSD and, besides talking to Puck, be capable of handling many other communication channels such as Internet, private fiber, satellite links, and HF radio. We will endeavor to make it secure but recognize that with the necessary added complexity comes added risk. So Broker holds mild secrets, enough to deter most wiretapping or impersonation, but nothing crucial to hotline message content.

In CATALINK we believe we can achieve radically better simplicity because we are not trying for ultimate performance. Hardware has gotten orders of magnitude faster since the '60s era Moscow-Washington hotline that we're reconstructing and we're willing to spend some of that on simplicity. Not that performance doesn't matter; our final system needs to feel snappy. Perhaps even more important than hardware speeds, we believe an enabling factor for better security is that we have few legacy constraints.

Puck and Broker might live in the same office building and be connected by CAT5 cable through a Gigabit Ethernet switch. We assume a low bandwidth-delay product and a reasonably uncongested network. We do not entirely trust the network, though we are willing to assume that adversaries would find it challenging to disrupt. More specifically, from a security threat perspective, we feel the need for encryption to protect against wiretap or forgery but do not feel the need for extraordinary defense against Denial-of-Service packet flooding or against physical disconnection. To the extent feasible, we would like any network outage to be diagnosable by relatively unskilled staff, who can spot unplugged or frayed cables or power failures, but not DNS

misconfiguration. Initial setup may require an expert, as would ongoing systems administration of the Broker.

The [Precursor](#) device uses WiFi rather than wired ethernet but the security relevant network stack is a COM bus that is also designed for simplicity, so we're willing to switch to that if project hardware choices dictate.

It is likely that an implementation will deploy multiple Brokers (think one in the White House and one on Air Force One) exchanging state frequently. For messages to get through it would be enough that the Puck be able to talk to any one of these Brokers. For initial clarity, assume a single Broker.

The current state of the proposal is aimed at prototypes on common platforms but prepares for a tiny OS on future devices, and therefore we imagine a primitive UDP interface provided by the operating system and limit what we ask of it to handling the network hardware and multiplexing packets (based on UDP port number) to and from independently scheduled user-mode applications which are responsible for their own retransmission and encryption. We do not ask for DNS or routing configuration or the rest of a normal IP infrastructure.

puckfs (For historical origins, see [Plan9 rudp](#).)

Instead of general message passing, for more understandable control flow we conjecture all we need is remote procedure call. Furthermore, rather than general RPC, we think we can get by with a primitive remote filesystem, reading and writing full files. For simplicity, we decided:

- no walk, always use canonicalized pathname
- no stat, that info comes when reading directory
- no mkdir; directories are created as needed when file is written
- no symlinks, no hard links, no locking, no ACLs

The **client** calls

```
p, err = puckfs.Dial(secretFile)
data, err = p.ReadFile(path)
err = p.WriteFile(path, data)
err = p.Remove(path)
fi, err = p.ReadDir(path)
err = p.Chtime(path, mtime)
err = p.Close()
```

Here `secretFile` is a local file containing JSON fields: `RemoteAddr`, `MTU`, `KeyID`, `Secret`. `KeyID` is unique to a (client,server) pair and is used in both directions. Calls are synchronous. The filesystem is single-user, so has no permissions or access control beyond the authenticated encryption of the network link.

The **server** process is deliberately primitive, supporting a single remote user and running as a custom local `userid` with access to just the one dedicated directory tree.

If we need more functionality, try files first. For printing, as an example, don't implement CUPS but WriteFile a PDF from the puck to a spool directory on Broker. It seems highly likely that the principal creating and responding to messages on the Puck may wish to include text and photos from staff. Again the puckfs mechanism may be helpful via relay to legacy network file systems.

FAQ

Q: What is the user interface?

A: Since Puck hardware is still to be settled, discussion of UI is somewhat premature. For this proof-of-concept implementation, we're building on top of OpenBSD and running commands within the acme text editor. Error messages are seen directly on that screen.

Q: Why the belt-and-suspenders construction of nonce in the implementation?

A: Crypto-random bytes are a well-accepted way to implement a nonce, so we wanted to use that for sure. To ease concerns about subverted randomness generators, it was mildly attractive to also have an independent means of assuring the nonce is unique. There are presumably other bad things cryptographically that happen if your random source isn't. So take pains to check that.

The main reason to have monotonic counters with fatal errors is as a tripwire for otherwise undetected theft of the secret. I acknowledge there may be operational pain in keeping global counters in sync, even just between Puck and Broker, so might have to revisit whether this is treated as a fatal error but will try for now.

There is intended to be no downside to using both. As long as either mechanism is working, even if the other is controlled by an adversary, the nonce should be safe. If not, that's a critical bug.

Q: The rest of the puckfs is minimalist; why include Chtime?

A: Yes, it was a borderline decision. Often enough we want to change file format, say compressing in place, without losing track of which semantic version of the file is involved.

Note that we gloss over the problem of native filesystem time resolution. Since we mainly copy times of existing files, the subtle rounding is often negligible.

Q: Why not adopt the FS abstraction of Go 1.16?

A: `ReadDir` returns a `[]fs.FileInfo`. We don't need the optimization introduced by distinguishing `DirEntry` from `FileInfo` and we do need file write. So the new abstraction didn't seem a particularly good fit here, but perhaps I just don't understand; corrections welcome. Anyway we're aiming for a design that is bland and not tied too intimately with Go. With luck the average reader will not know or care about `fs` versus `ioutil`.

TODO

We must provide a safe way to set (and reset) the shared secret between client and server. PAKE seems a possible choice, but let's get the rest settled first.

We want some **notification** channel, for example to be alerted when a directory changes. We also want something similar to **mossh**.

Implicitly, the intent here is to standardize the wire format of `puckfs` as well as the client and server API. Such a standard makes it easy to later adopt different languages for implementing Puck and Broker. It was tempting to adopt 9P2000, but that seemed overkill for our needs.

[AdamWick] I believe you may want multiple tools to consider different parts of the system. In particular, there are four possible things you might want to prove:

1. That the designed puck-to-puck crypto protocol maintains the integrity and confidentiality of the data. Since you're not actually trusting the puck-to-broker or broker-to-broker channels to maintain C/I, proving stuff about them isn't nearly as important, but puck-to-puck needs to be proven. I believe that the state of the art there is automated theorem proving (Coq, Isabelle, etc.) via crypto games, but I'm not an expert. I do know a few, however, if we want to bring one in at some point.
2. That the puck-to-broker/broker-to-puck protocol doesn't have any weird corners; more formally, you probably want a couple "always eventually" properties, like "the message from the puck always eventually makes it to the broker if it is physically possible." SPIN is still around for this sort of thing. I've also heard of people using Ivy, TLA+, P, and Alloy these days. Some of the newer tools can also generate test vectors to test your implementation, too. (Although, of course, it'd be lovely to verify it, too.)
3. That the broker-to-broker protocol similarly doesn't go into a corner. Same as #2, perhaps, but it depends on how you design things. Again, you're probably looking to prove properties around "at most once" delivery, since you can use your monotonic counter to clear double-deliveries. Also, if the broker-to-broker protocol goes full mesh (which, given the context, is probably wise), those protocols are often hard to reason about in a dynamic universe. Even if I pulled something off the shelf, I'd want to get a good formal grounding for it given whatever fundamental technologies are used. Dynamic routing across the world is hard!

4. That the implementation of the puck's crypto and messaging actually implement the design you came up with in #1. For this, I will recommend Galois's SAW and Crux tools. :) We have some partial support for Go currently, although it hasn't been used in anger (to my knowledge). [Hah: As I typed that I got an update: "... [it] can be experimented with ... While you shouldn't use it in anger, attempting to do so would at least probably make you angry?"] We could also run through `gollvm`, as our tools will process LLVM bitcode; I'm currently doing this in a DARPA project to try to verify some Rust code, which also compiles through LLVM. Either way, there's some effort to get the tooling to work nicely with Go, but in neither case are we starting from scratch. [/AdamWick]

I thank Dave Presotto for comments on this proposal and even more for teaching me all this in the Plan 9 days. I thank Alexa Wehsener and Leah Walker for clarifying that ROCCS refers to the network; Broker is the proxy device. Adam Wick added some great suggestions on which formal methods may be most relevant.