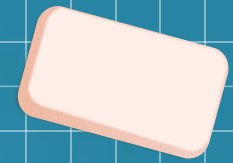
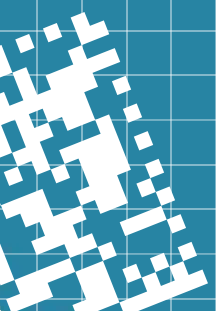
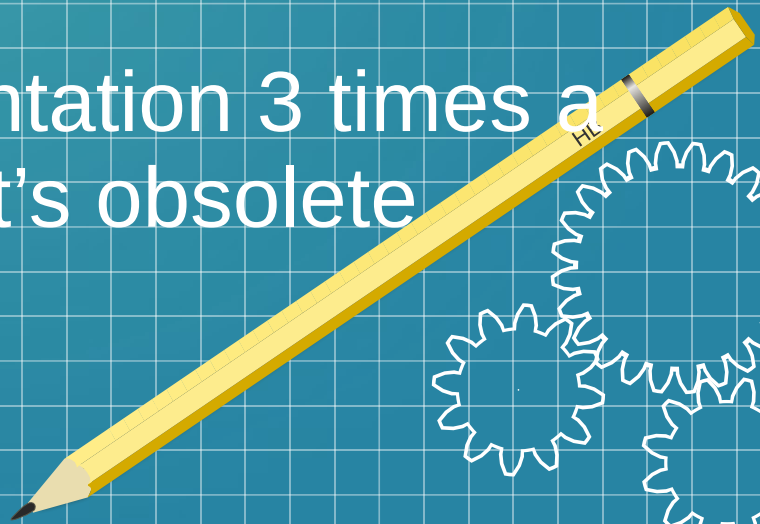
Embedded I2P: The Final Chapter

Or: How I gave the same presentation 3 times a year for 8 years, and now it’s obsolete







Welcome CCC!

- Presenters: eyedeekay, Aaro Altonen
- eyedeekay: Current Java I2P co-maintainer, author of go-i2p and go-i2p-* userspace
- Aaro Altonen: Independently implemented I2P in Rust

What do we mean when we say ‘Embedded I2P’

- I2P routing services can be `Embedded` in an I2P Application
  - This has been challenging in the past with Java and C++
  - In Java the limitation is clear and simple, to embed Java I2P into an application, that application must run on the Java Virtual Machine(or at least, something like it, Dalvik for instance)
  - In C++ the limitation is a little more specific and has to do with compiler behavior(Name Mangling). In C++, one must define a C interface which wraps the C++ interface. This is rarely as natural as it should be.
  - So “Embedded I2P” usually consists of a statically-compiled I2P router running in something more easily recognized as an I2P application.
  - The most well-known “Embedded I2P” app at this time is BiglyBT

Does that mean we have to have a router in every relevant language?

- No, of course not.
- But it does mean we need interfaces that work across languages.
- We have some interfaces that work across languages: SAMv3.3 is our flagship socket-like API for building I2P applications. There is also i2pcontrol for obtaining information from the router at runtime.
- None of the existing APIs are capable of end-to-end lifecycle management of the router itself.

SAMv3.3

- SAMv3.3 is an API for configuring I2PSockets from an external program.

```
HELLO VERSION MIN=3.1 MAX=3.1

Needed for all of the remaining ones

DEST GENERATE SIGNATURE_TYPE=7

To generate our private key and destination

NAMING LOOKUP NAME=...

To convert .i2p addresses to destinations

SESSION CREATE STYLE=STREAM ID=... DESTINATION=...
i2cp.leaseSetEncType=4,0

Needed for STREAM CONNECT and STREAM ACCEPT

STREAM CONNECT ID=... DESTINATION=...

To make outgoing connections

STREAM ACCEPT ID=...

To accept incoming connections
```

```
package main

import (
        "log"

        "github.com/go-i2p/onramp"
)

func main() {
        garlic := &onramp.Garlic{}
        defer garlic.Close()
        listener, err := garlic.Listen()
        if err != nil {
                log.Fatal(err)
        }
        defer listener.Close()
}
```

I2PControl:

- **I2PControl is an JSON-RPC API for querying information from the I2P router and executing configuration changes**

```
{
  "id": "id",
  "method": "Method-name",
  "params": {
    "Param-key-1": "param-value-1",
    "Param-key-2": "param-value-2",
    "Token": "**actual token**"
  },
  "jsonrpc": "2.0"
}
```

```
{
  "id": "id",
  "result": {
    "Result-key-1": "result-value-1",
    "Result-key-2": "result-value-2"
  },
  "jsonrpc": "2.0"
}
```

```
i2pcontrol.Initialize(*host, *port, *path)
_, err := i2pcontrol.Authenticate(*password)
if err != nil {
        log.Fatal(err)
}

message, err :=
i2pcontrol.Echo(strings.Join(flag.Args(), " "))
if err != nil {
        log.Fatal(err)
}
log.Println(message)
```

APIs are great. So what's the real problem?

- It all comes down to lifecycle management.

What does ‘Lifecycle Management’ Mean?

- Lifecycle management is a professional-sounding euphemism for us to tell you to install a freestanding I2P router and run it in the background.
- The problem is that process sucks. Like a lot it sucks.
- This is the install guide for a free-standing I2P router in Windows(eew, Windows, I know. We cant care right now, sorry FSF nerds. People use it). https://geti2p.net/en/download/windows
- The Windows install guide is 33 steps long. That is not onboarding, that is filtering. **This is a COLLOSAL barrier to entry, that expects considerable prior knowledge from Windows Users. That is extremely ill-advised.**

Why not i2pd?

- While i2pd is slightly easier to install on a host system, it requires even more prior knowledge to use than Java I2P.
- It lacks automatic updates and also lacks a way to inform users of updates.
- I2PD is written in C++, and while it’s incredibly competent C++, it’s still C++. Name-Mangling by the compiler still forces us to write and maintain a separate C interface. Interacting with any non C++ language requires elaborate configuration and at least a C compiler and a C++ compiler.
- The exception to the rule: If you’re already working in C++, i2pd is probably fine.

Embedded I2P Routers in Go

- We built go-i2p specifically to explore the embedded use-case.
  - Because Go is memory-safe and garbage-collected
  - Because Go has an easy-to-use, git-based `import` mechanism
  - Because Go has excellent support for coroutines and channels
  - Because Go has excellent I2P libraries
  - Because people want to rewrite the world in Rust, but I know Go very well and would rather use it

Embedded I2P Routers in Rust

- I2P in Rust is especially desirable
  - Because Rust emphasizes memory-safety and enforces it with the compiler
  - Because the Rust toolchain is extremely well-designed
  - Because people wants to rewrite everything in Rust, so the time for first-class I2P support in Rust is now
  - Rust downstreams want to use Rust upstreams
  - Excellent C Binding support

What changes?

- We can inform our designs with lessons learned from embedding Java I2P and i2pd
- We can provide developers with safe, transparent, and easy-to-use tools for lifecycle management for I2P applications

Cool Part 1: Simplest Embedded go-i2p Router

```
cfg := config.DefaultRouterConfig()
router, err := embedded.NewStandardEmbeddedRouter(cfg)
if err != nil {
    log.Fatal(err)
}

if err := router.Configure(cfg); err != nil {
    log.Fatal(err)
}

if err := router.Start(); err != nil {
    log.Fatal(err)
}
defer router.Close()

router.Wait() <--BLOCKING: This is where the I2P router
                Actually runs. Returns when you call
                router.Stop or router.HardStop
```

1. Create with NewStandardEmbeddedRouter()
2. Configure with Configure()
3. Start with Start()
4. Run with Wait()
5. Stop with Stop()
6. Cleanup with Close()

This is cool in and of itself, but we can be even cooler.

```
func NewGarlic(tunName, samAddr string, options []string) (*Garlic, error) {
        g := new(Garlic)
        g.router, err := embedded.NewStandardEmbeddedRouter(cfg)
        if err != nil {
                return nil, err
        }
        if err := g.router.Configure(config.DefaultRouterConfig()); err != nil {
                return nil, err
        }
        if err := router.Start(); err != nil {
                return nil, err
        }
        g.name = tunName
        g.addr = samAddr
        g.opts = options
        var err error
        if g.SAM, err = g.samSession(); err != nil {
                return nil, fmt.Errorf("onramp NewGarlic: %v", err)
        }
        if g.StreamSession, err = g.setupStreamSession(); err != nil {
                return nil, fmt.Errorf("onramp NewGarlic: %v", err)
        }
        return g, nil
}
 func (g *Garlic) NewListener(n, addr string) (net.Listener, error) {
        while {
                // Wait until the SAM API is ready...
                if _, err := net.Listen(tcp, “localhost:7656”); err != nil {
                        break
                }                               func (g *Garlic) Close() error {
        }                                               if err := g.router.Stop() {
        listener, err := g.Listen(n)                            return  err
        if err != nil {                                 }
                return nil, err                         if err := g.router.Close(); err != nil {
        }                                                       return err
        return listener, nil                            }
}                                               }
```

Cool Part 2: SAMv3 API with Embedded I2P Router

1. Create with NewGarlic()
2. Router “Starts” automatically with NewGarlic call, and prepares a SAMv3 API
3. Listen for I2P connections with listen
4. Close SAMv3 connections with Listener.Close(), shutdown router normally

Now, even the developer of the application usually does not have to think about how to enable embedded I2P.
It “Just Works.”
TM.

What does it all mean?

- For developers:
  - Adapting your application to I2P becomes easy, all the underlying work can be done automatically
- For users:
  - Using I2P becomes zero-config, in fact it become nearly invisible. Long onboarding processes are a thing of the past
- For I2P:
  - By helping developers use I2P with correct lifecycle management, we grow the I2P network while improving network stability and performance

Future?

- Go has a module dependency management interface which allows inline replacement with the `replace lib => otherlib` directive.
- Could we create our own version of the `net` namespace to enable I2P applications with NO CODE AT ALL? Probably

replace “net” => github.com/go-i2p/net

For this still hypothetical example, `go-i2p/net` manages the I2P router lifecycle, which terminates with the application itself.