[

Untitled-2024-08-27-1129

2664×1932 454 KB

](https://europe1.discourse-cdn.com/flex013/uploads/anoma1/original/1X/00e62686dacf3398d079a3a8e09c6e782e22aa14.png)

This is the physical transport engine design.

An Erlang system is a maximum trust domain, so messages between engines on the same Erlang system are just simple Erlang messages. We never rely on Erlang for security; this is all arbitrary code running on our CPU (even if we use Erlang distribution, this is just system administration).

So every engine has an Erlang PID (it's actually a process group, but it has one PID that receives interengine messages). We extend this to foreign engines as well, creating a process for each foreign engine we know about, and registering it under that address just as with local engines.

The difference is that these foreign engine proxy processes are not the actual engine. Instead, they do two things:

1. Sign, MAC, and whatever the message with the local sending engine's key; this involves asking the keyholder (currently Router) to do the signature.

2. Forward {src, dst, signed_message}

to wherever it's been told to forward signed messages to. This could be a TCP connection process (holding a physical TCP connection), a Unix socket process, or anything. It's just a PID.

On the receiving side, the TCP connection process verifies signed_message

, silently dropping the message if this fails, and sending it as a normal Erlang message to dst

if it passes.

Transport is the supervisor over these two types of processes (foreign proxies and physical connection holders). It also supervises e.g. TCP listening sockets and similar, which create physical connection holders. Upon the creation of a physical connection, we simply burst "learn this engine" messages for all our engines, signed by Router whose job is knowing what "all our engines" means. Over its lifetime, we may send new "learn this engine" messages and also "forget this engine" messages; these create and destroy foreign engine proxies on the other side.

The practical reality of this is that a message sender need do nothing special, ever. It simply sends the message it wants to send as an ordinary Erlang message, it's just that the recipient may be the above-described transport which delivers to the actual engine instead of the actual engine. The built-in Erlang global

registry may be used effectively here; it maps engine names to PIDs and its names may be used as arguments to e.g. GenServer.call

and friends.

This design smoothly handles call and cast trivially, but subscription needs additional design work. Trivially, each proxy can subscribe to the firehose and this would make porting the semantic trivial, but we don't want or need that much network traffic. As such, engine proxies should have a specific filter spec as part of their state, which they subscribe to; what this actually contains is a matter for future design. Most will be able to subscribe to nothing.