Assemblages: Modules with Interfaces for Dynamic Linking and Communication

Yu David Liu and Scott F. Smith

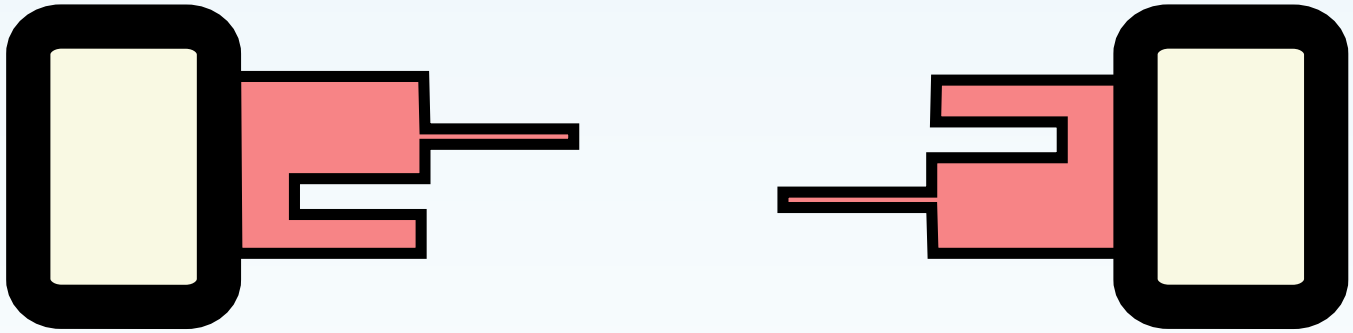
The Johns Hopkins University

The Main Design Principle

Every form of interaction of code with the environment should be supported by explicit, declared interfaces.

A Typical Module

Examples: Functors, Mixins



Modern Language Interactions

Main forms of interaction include

- Code-code interaction via static linking
- Code-runtime interaction via dynamic linking
- Runtime-runtime interaction via distributed messaging

Our Approach: Assemblages

- A module-centric approach to interfaces for all three
- Static linking of assemblages—mostly standard stuff
- Main contributions:
  - Module-centric view of dynamic extensibility and distributed messaging
  - Unified treatment of all three, using related notions of interface

Code and Runtime

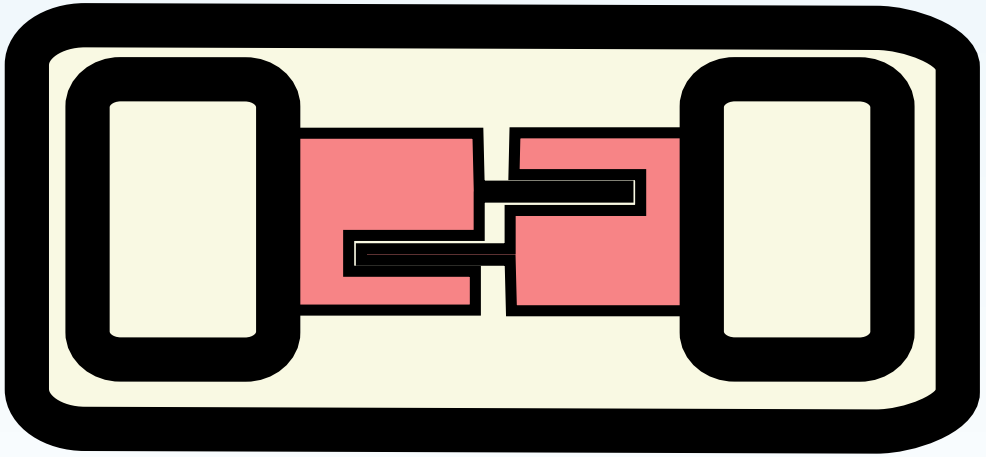
- *Assemblages* are code units—modules
- *Assemblage runtimes* are loaded assemblages

Static Linking Illustrated

Assemblages with static linkers being linked together:

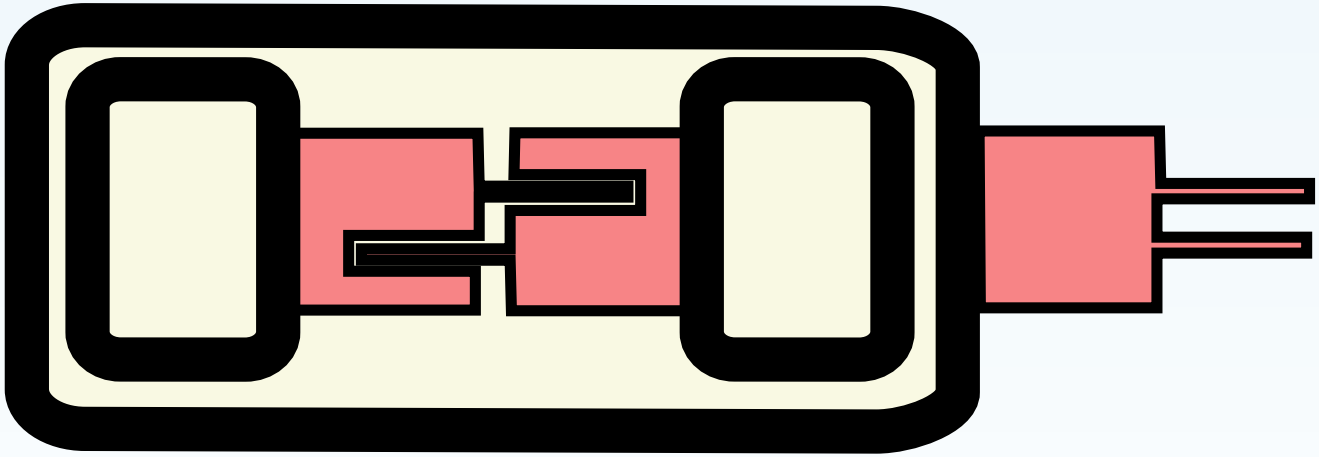
Static Linking Illustrated

Assemblages with static linkers being linked together:

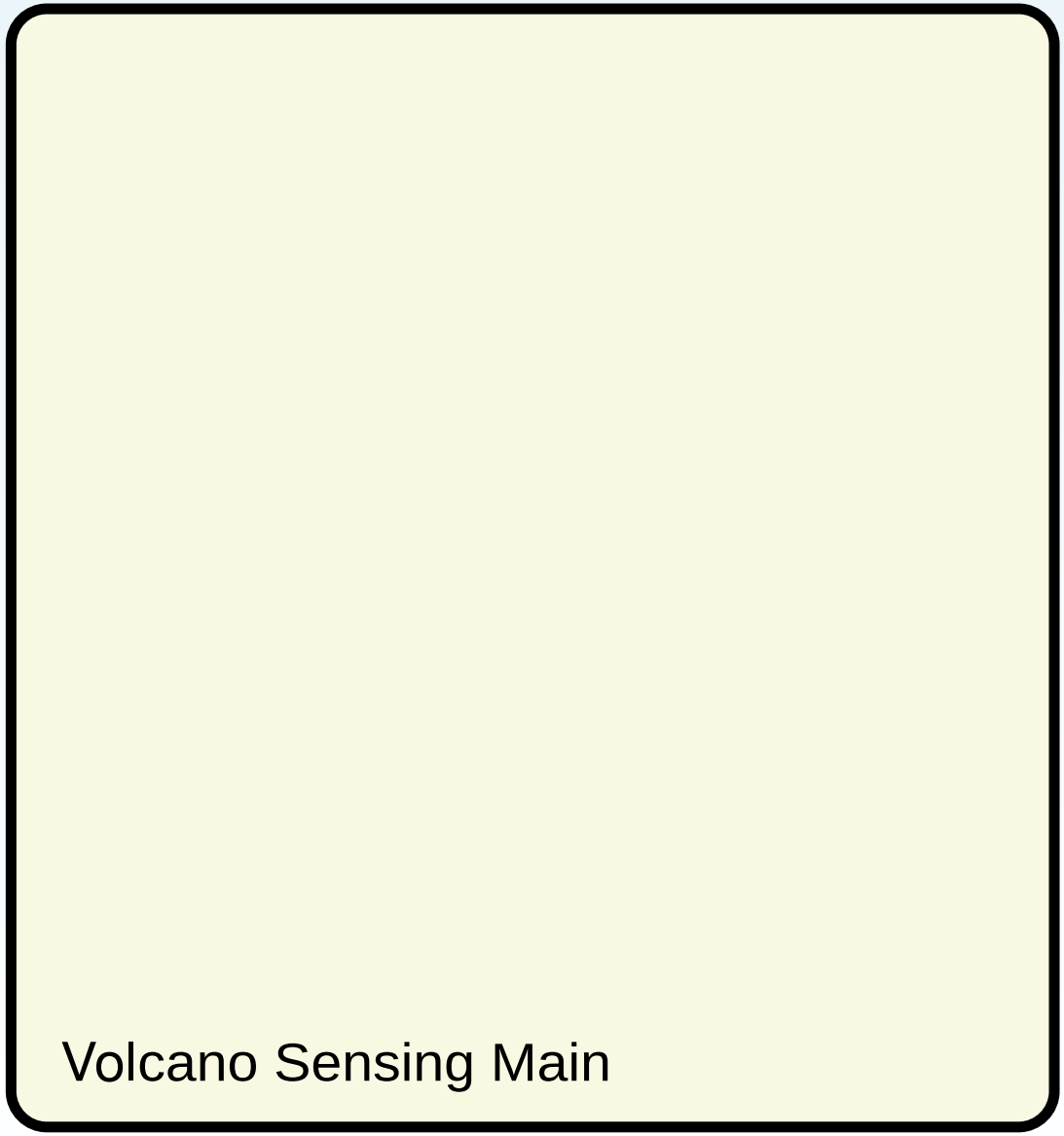


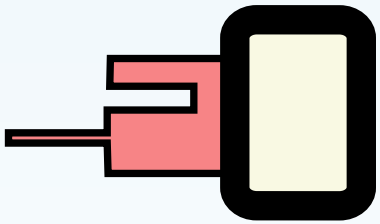
Static Linking Illustrated

Assemblages with static linkers being linked together:



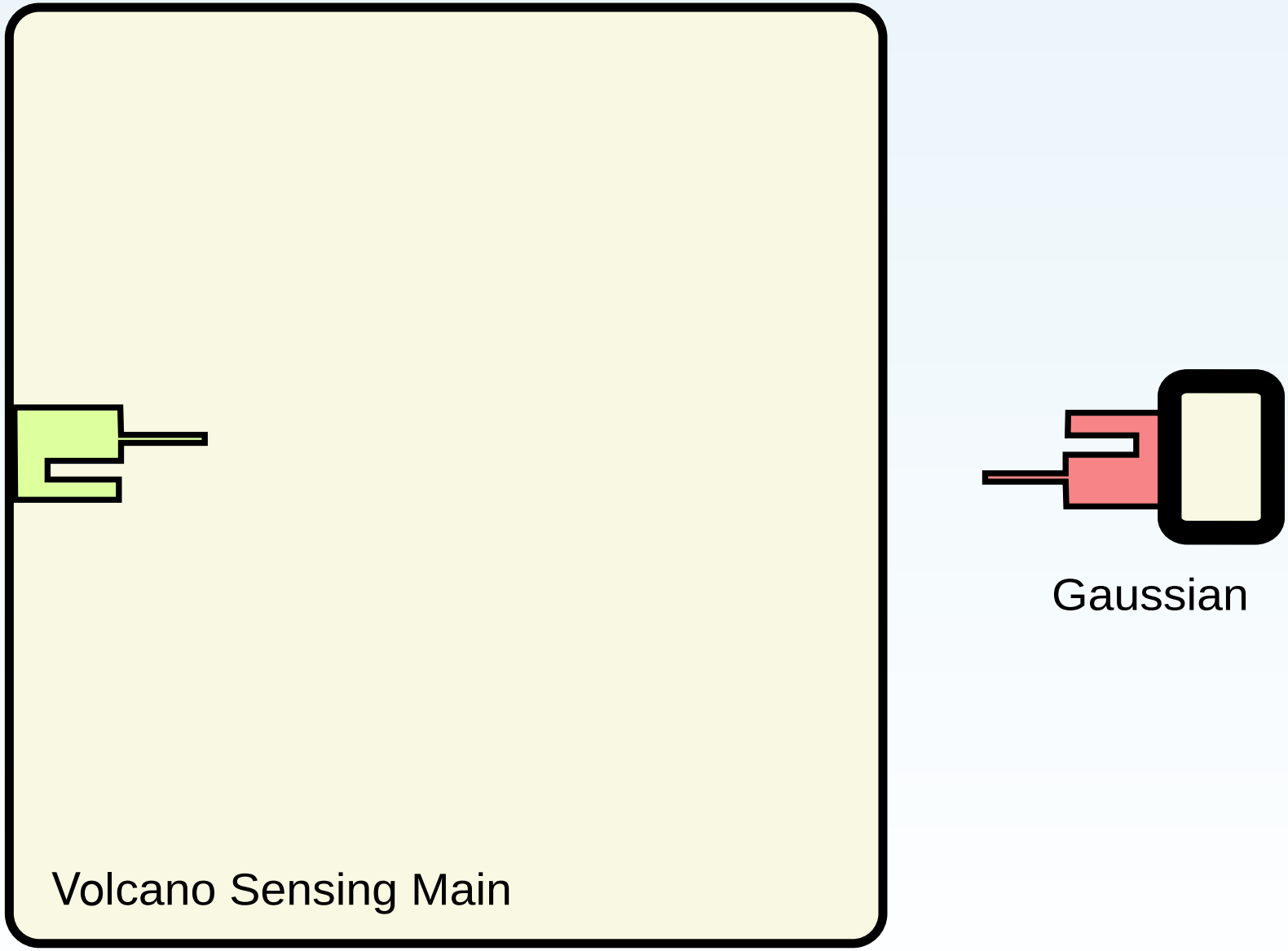
Dynamic Linking



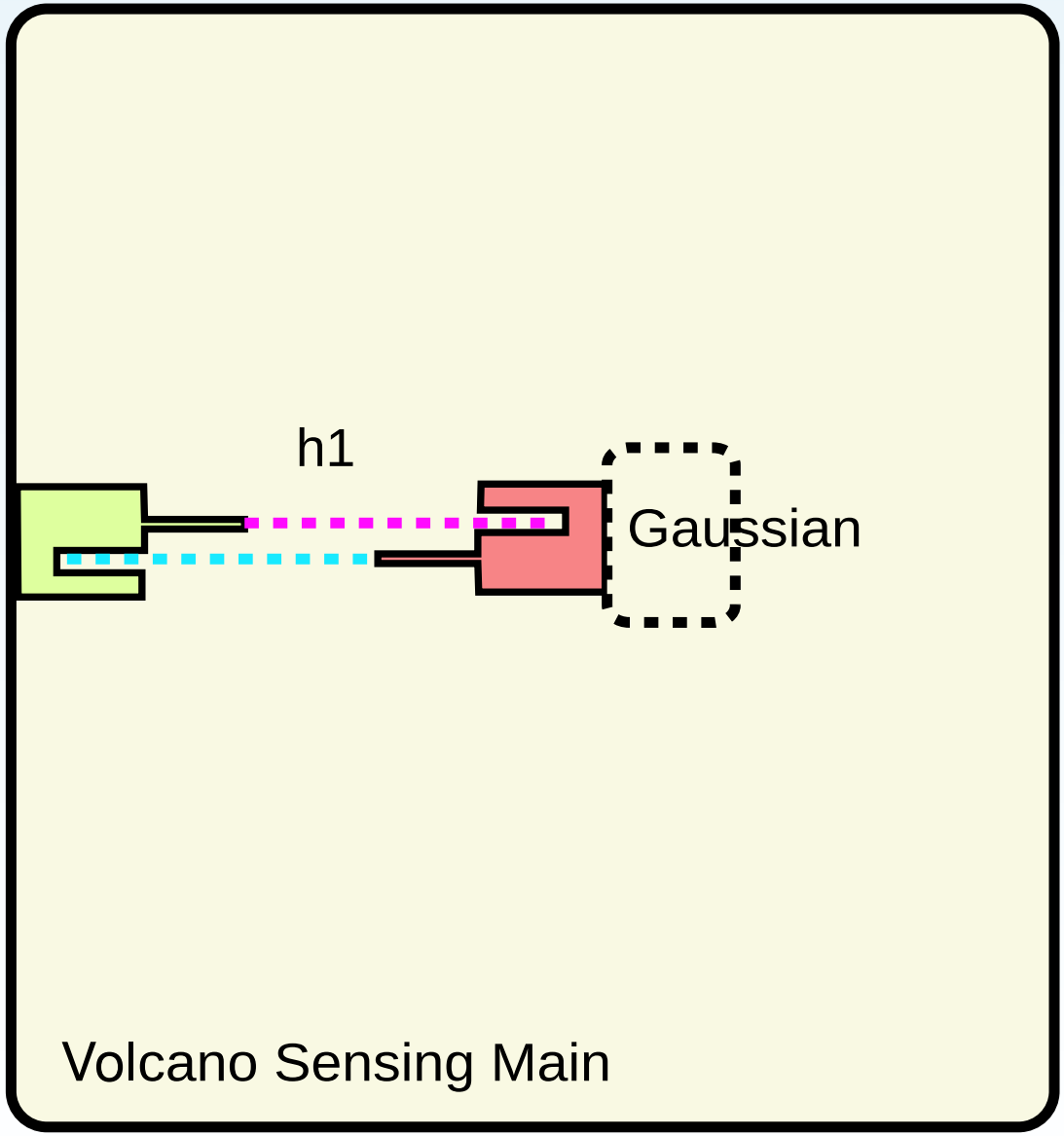


Gaussian

Dynamic Linking



Dynamic Linking

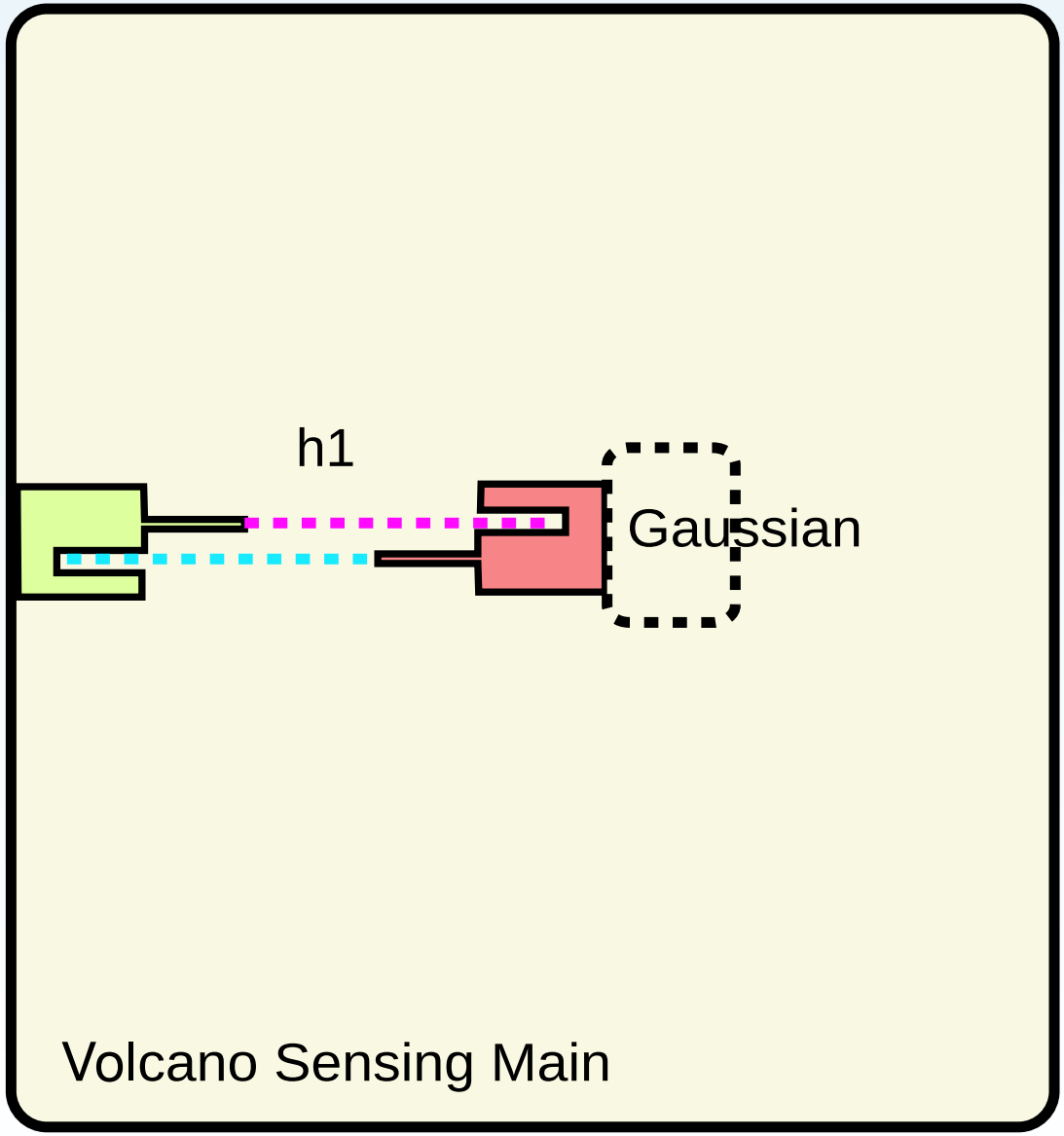


Rebindability of Dynamic Linkers

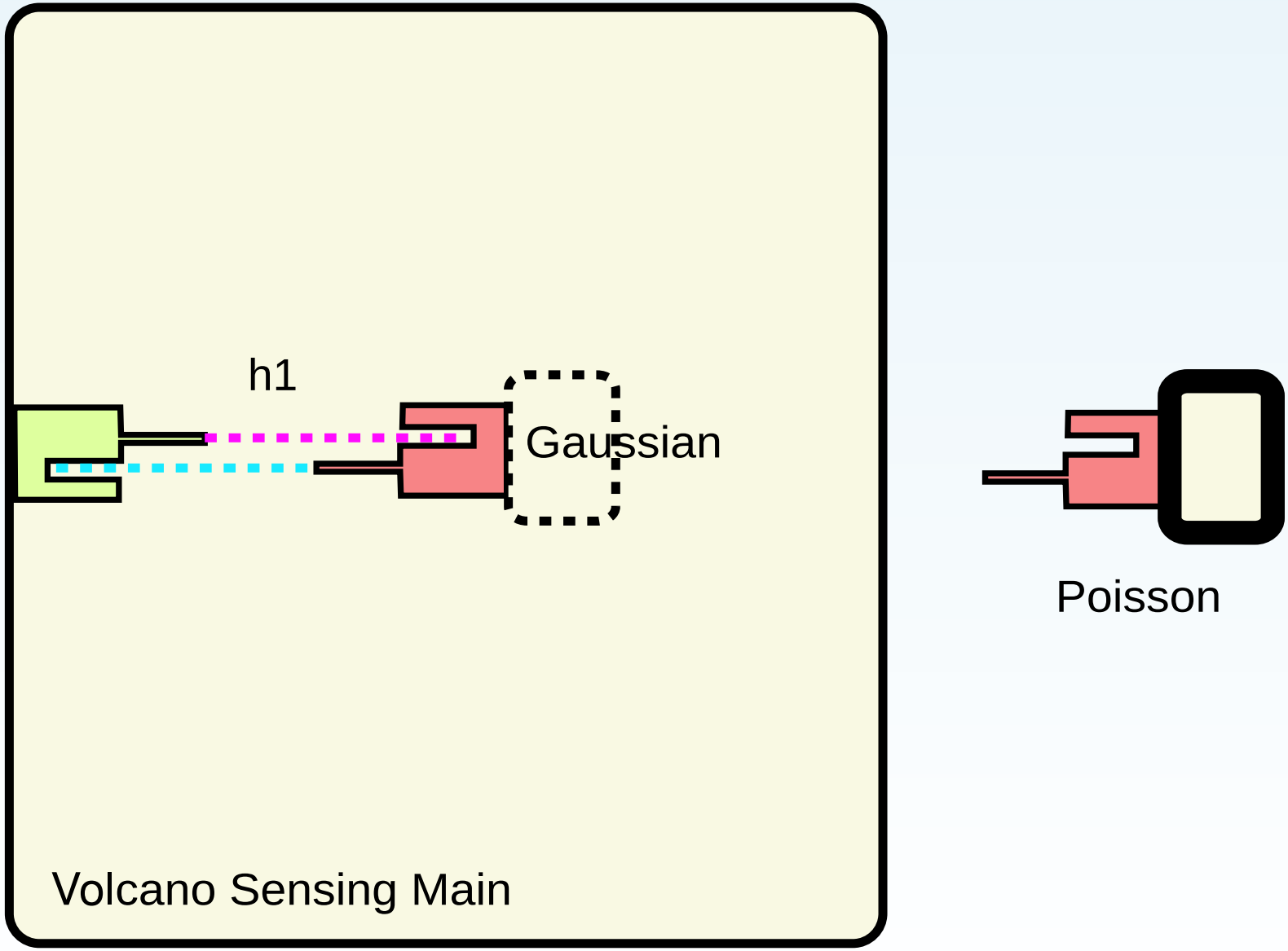
- Problem: The need to allow *arbitrary* code extension
  vs
  Fact that only a *fixed* number of dynamic linkers can be on an assemblage

- Solution:
  - Allow *multiple simultaneous* linkings on a single dynamic linker
  - Get a different name ("handle") to access each one
  - Unlinking is implicit via garbage collection when handle is unreachable
  - Use of handles also avoids an “unlinked” error – if you have a handle, it references a real plugin

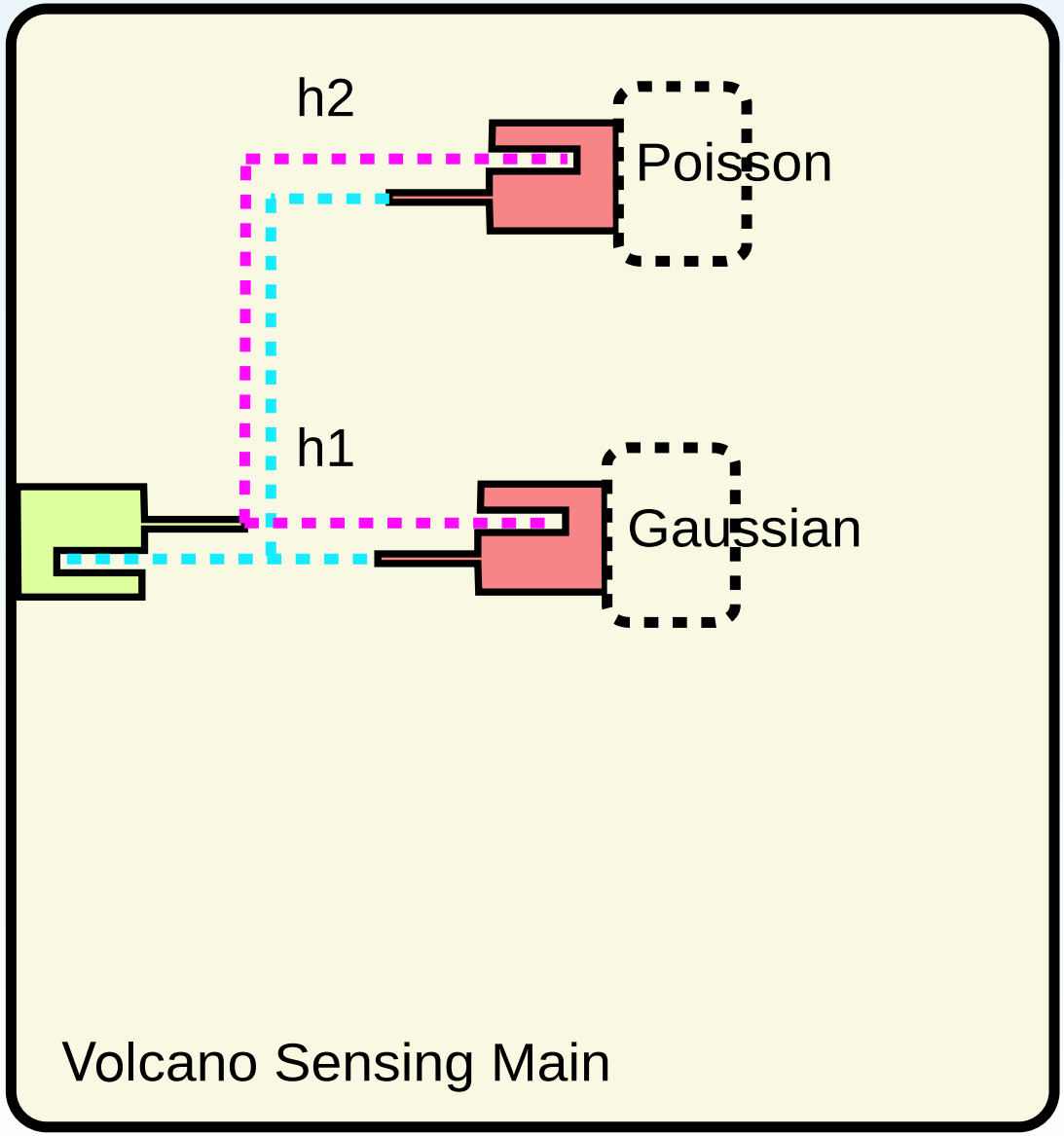
Rebindability illustrated



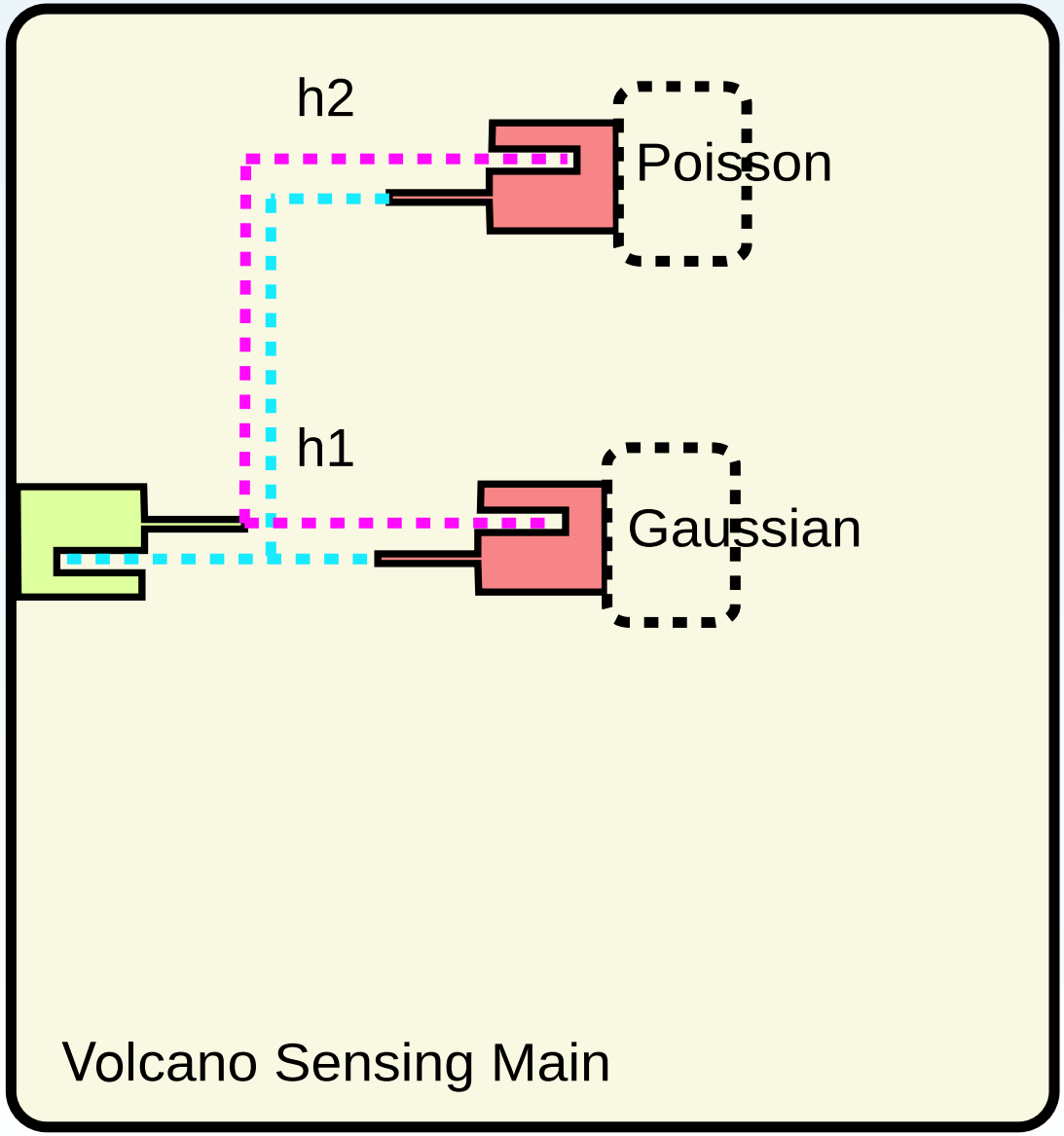
Rebindability illustrated

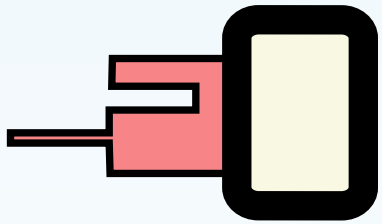


Rebindability illustrated



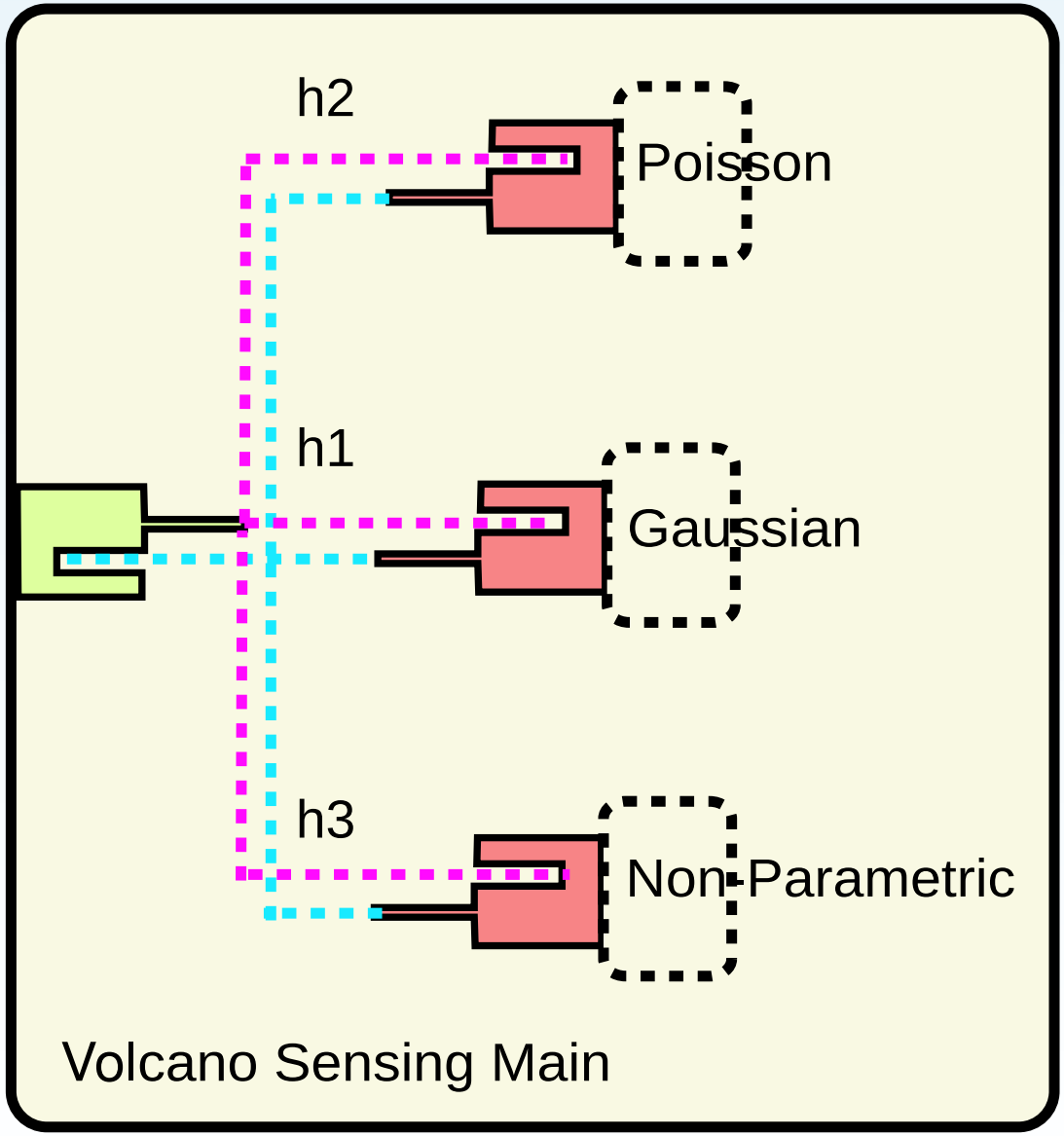
Rebindability illustrated



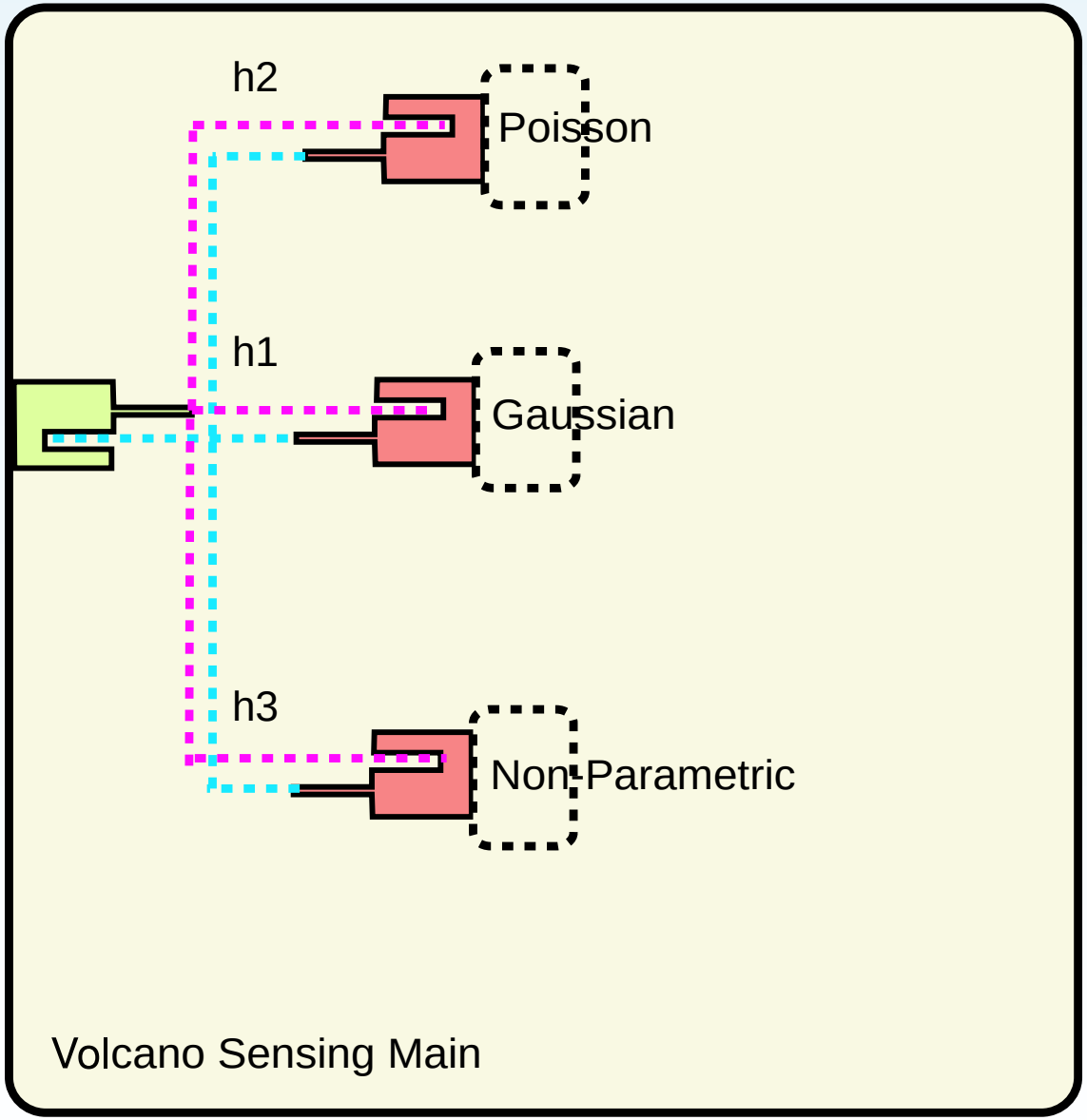


Non-Parametric

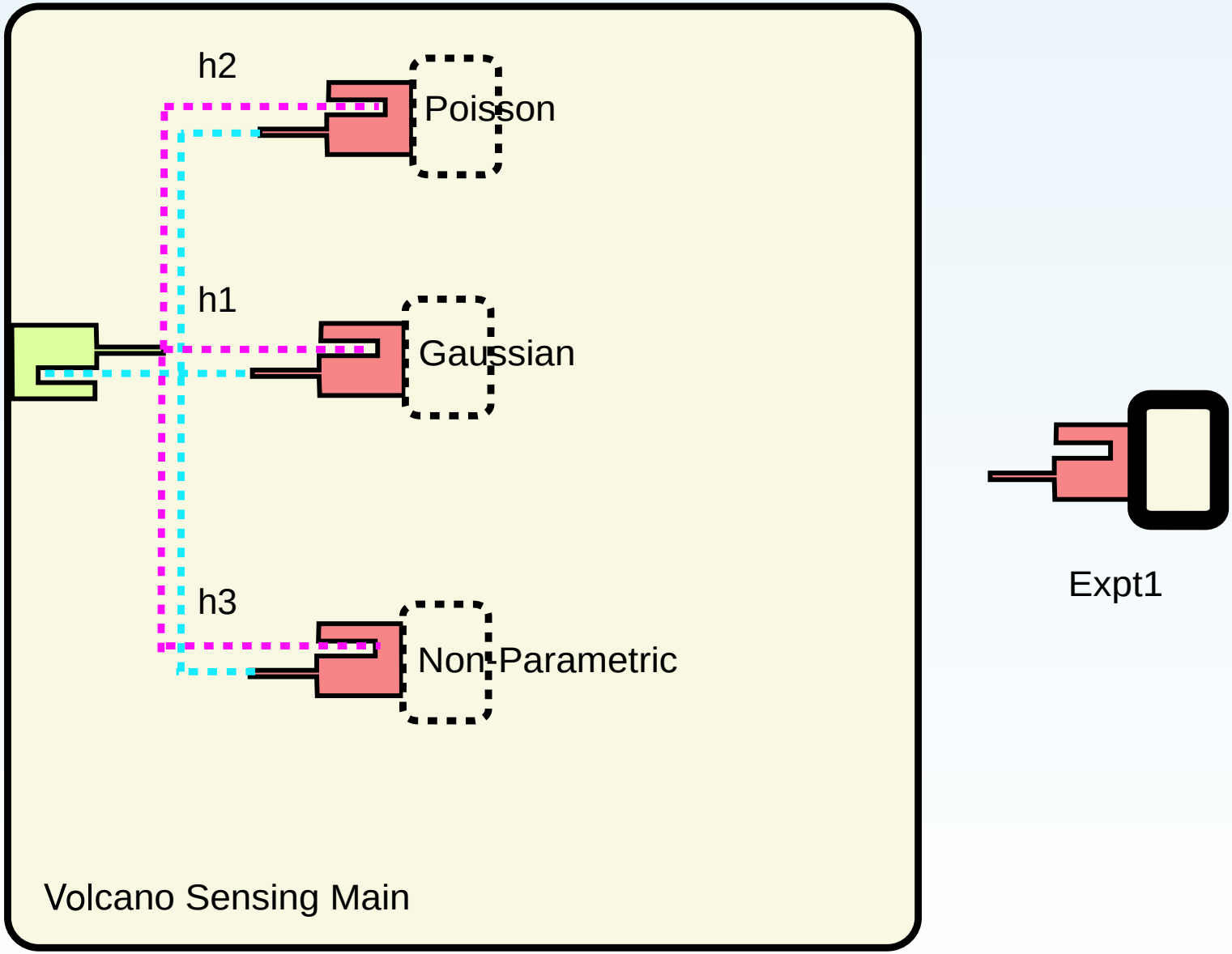
Rebindability illustrated



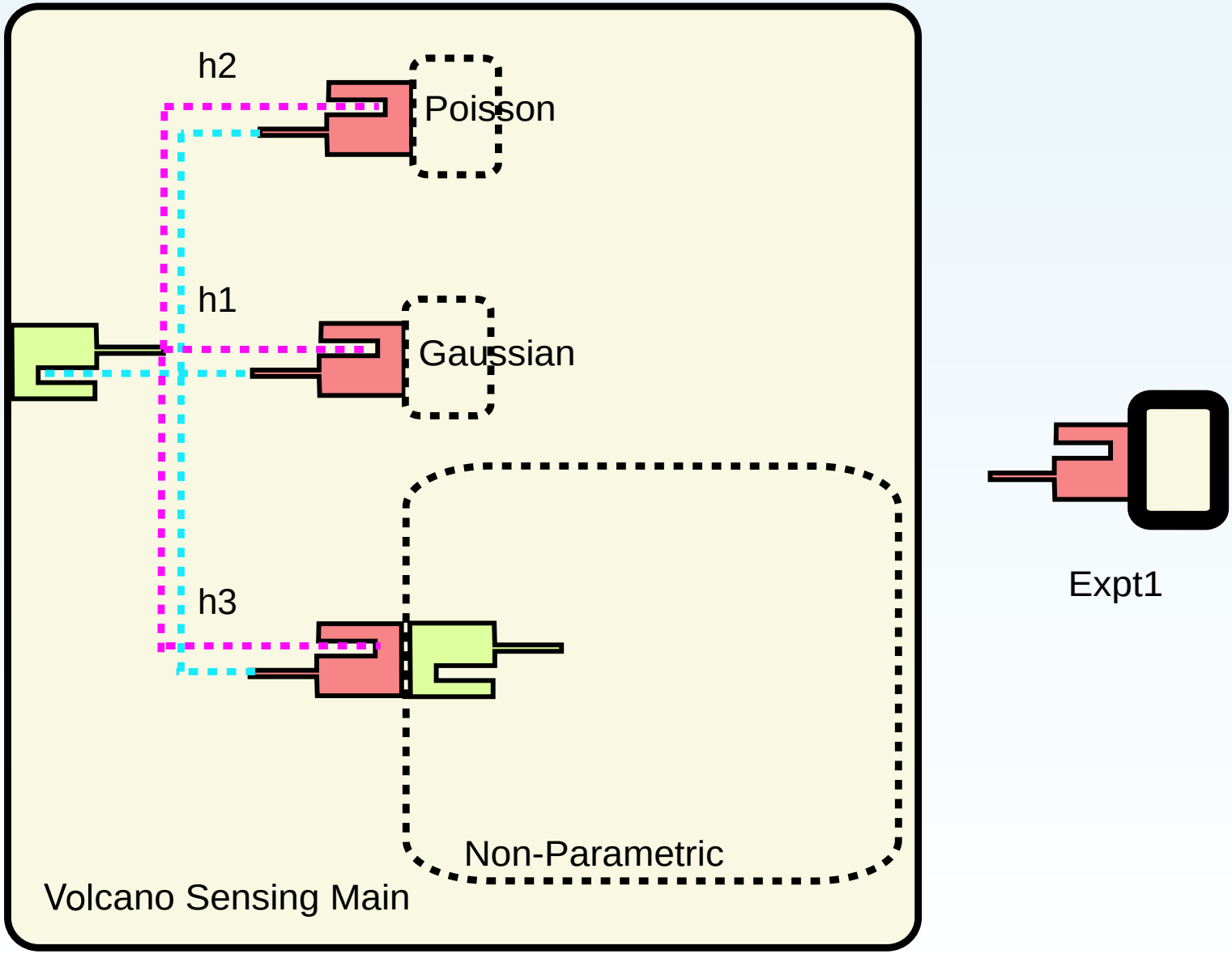
Dynamic Linking Tree



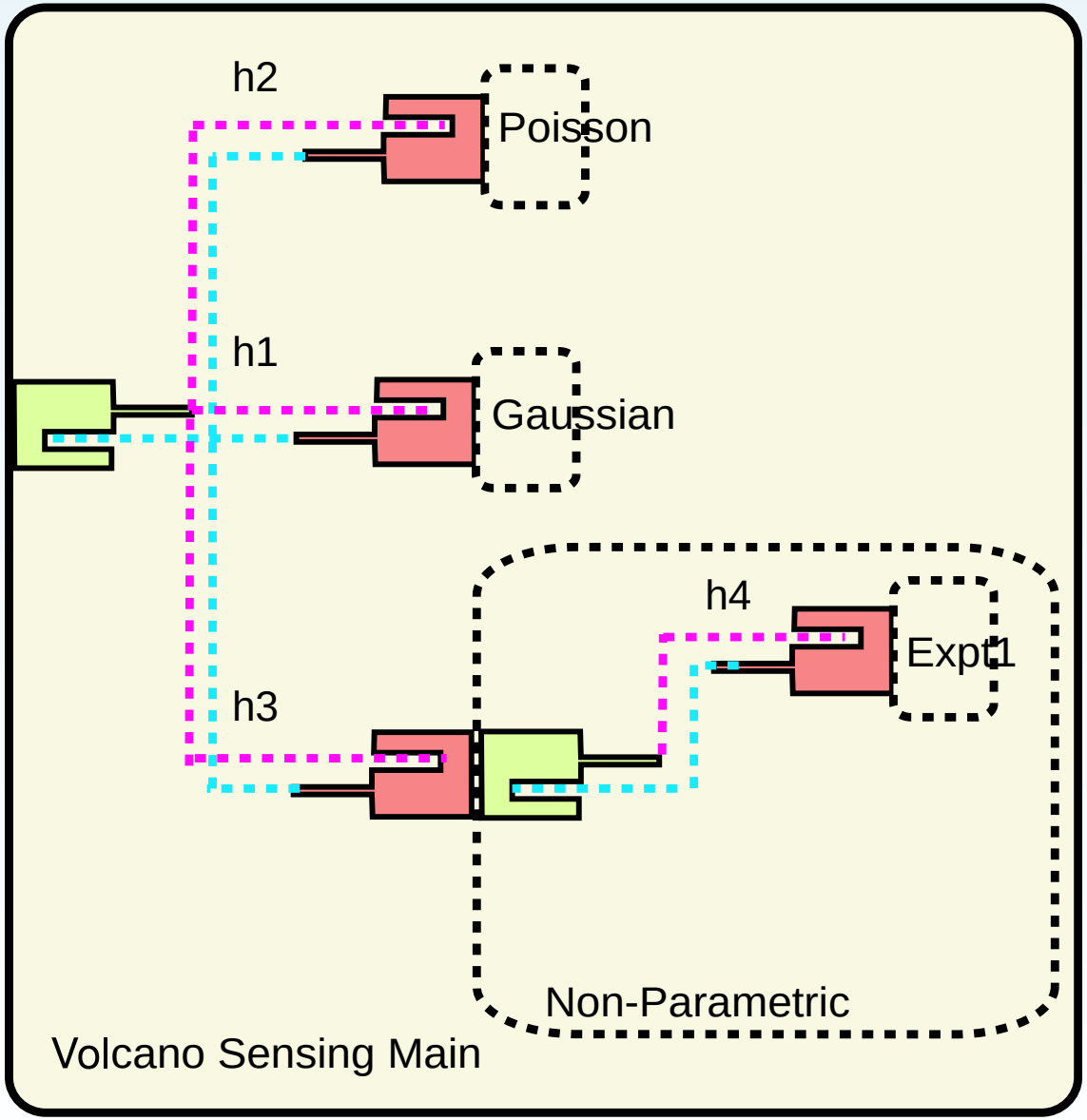
Dynamic Linking Tree



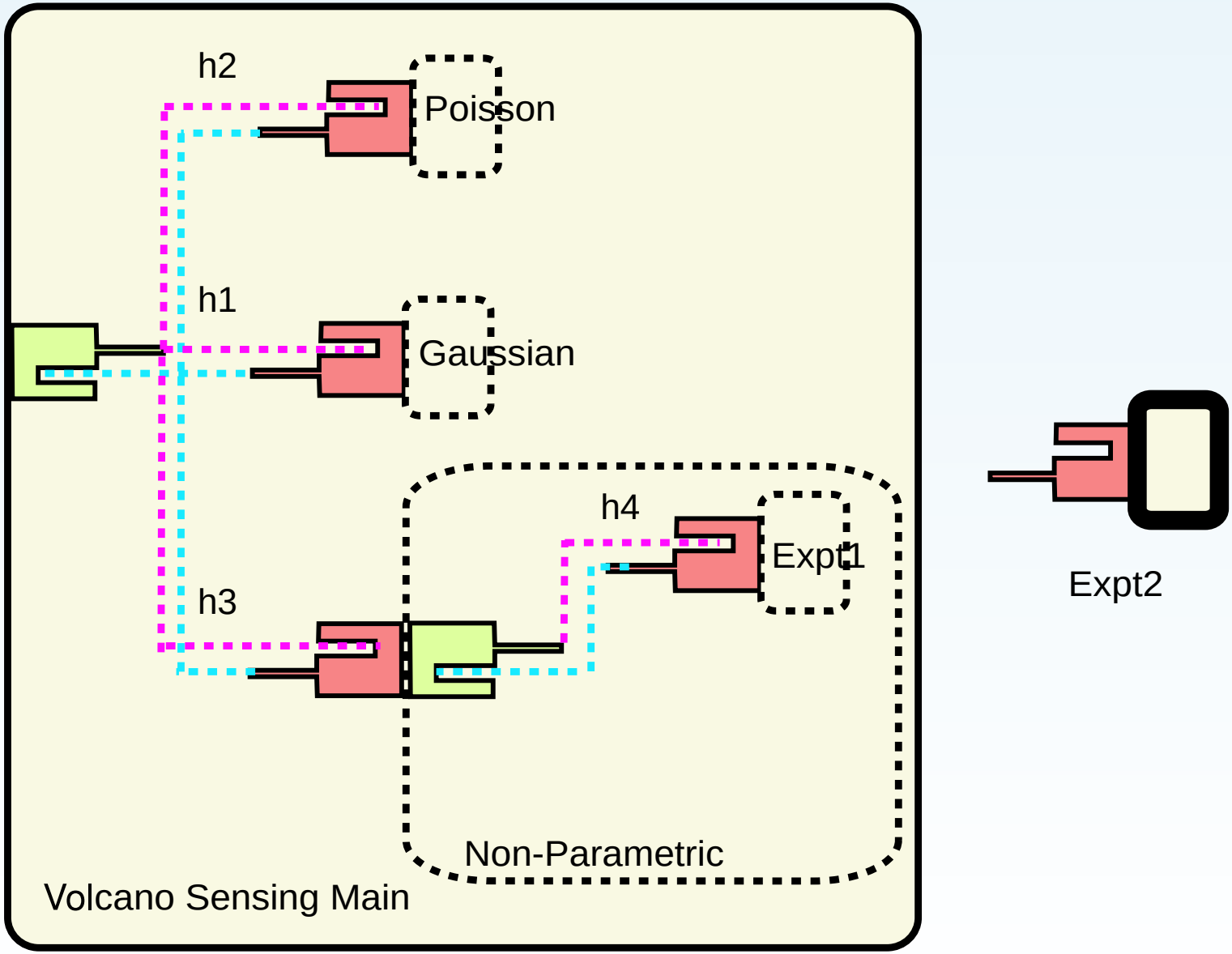
Dynamic Linking Tree



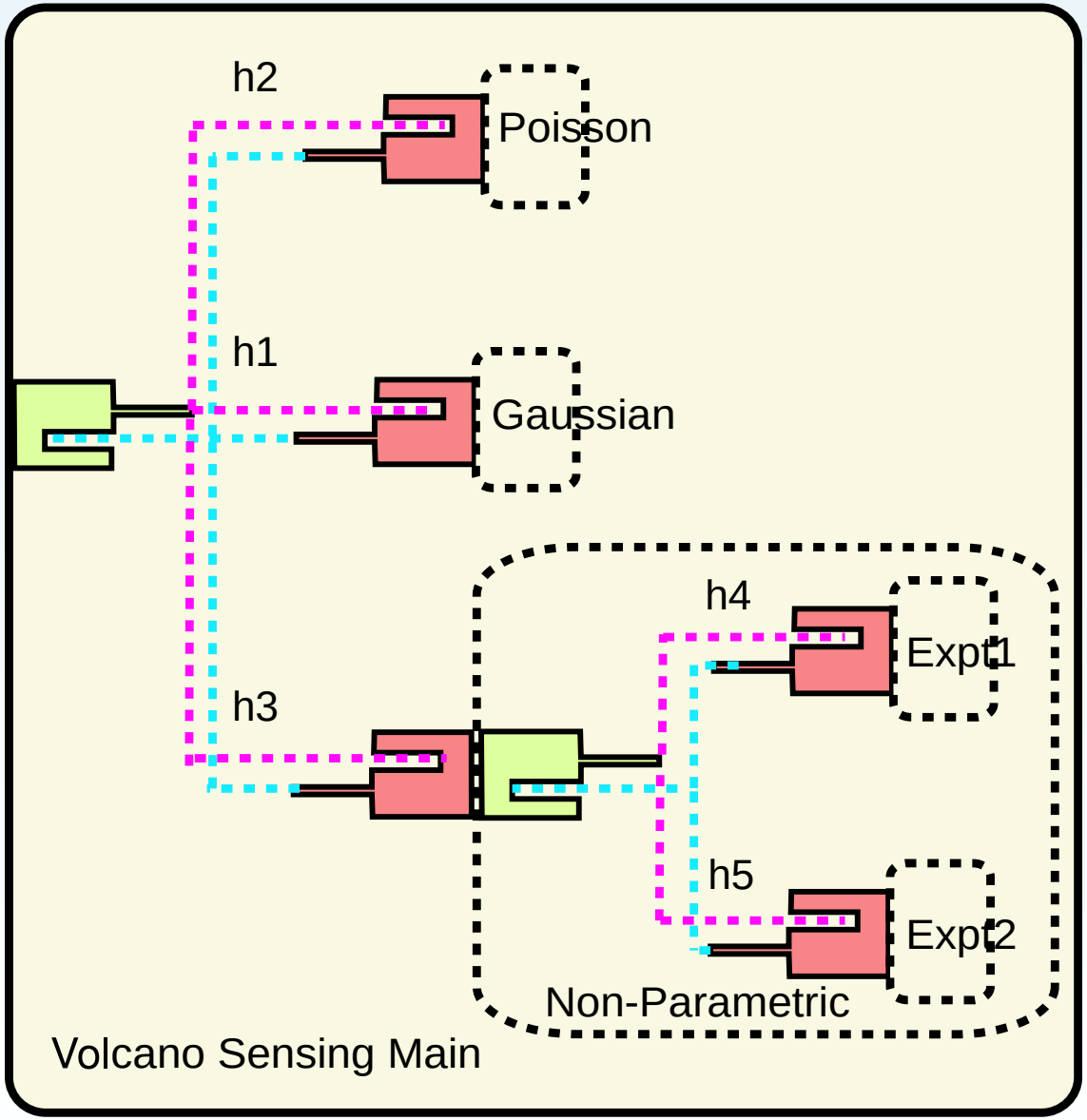
Dynamic Linking Tree



Dynamic Linking Tree



Dynamic Linking Tree

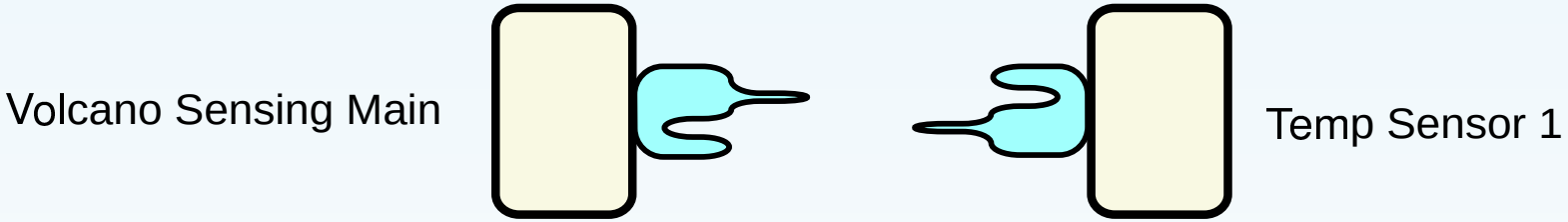


Connectors for Messaging

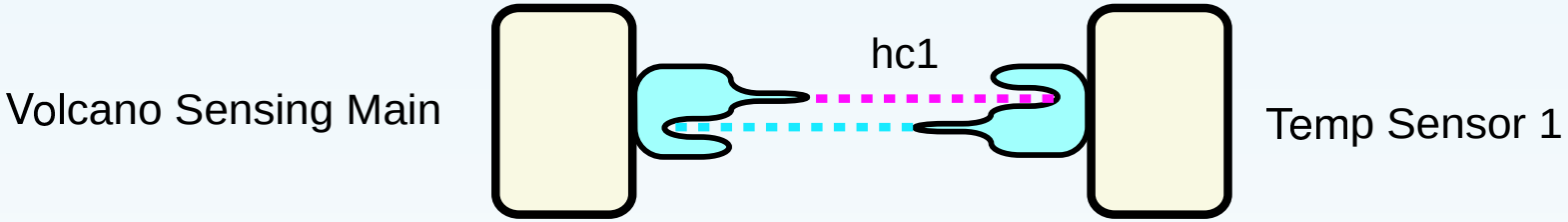
- Connectors allow different assemblage runtimes to interact
- Aim is a cleaner notion to replace RMI/RPC
  – RMI/RPC lacks a declared interface of interaction
- Connectors in programming languages are not completely new
  - Previously used in e.g. Cells and ArchJava and Darwin

Distributed Messaging Illustrated

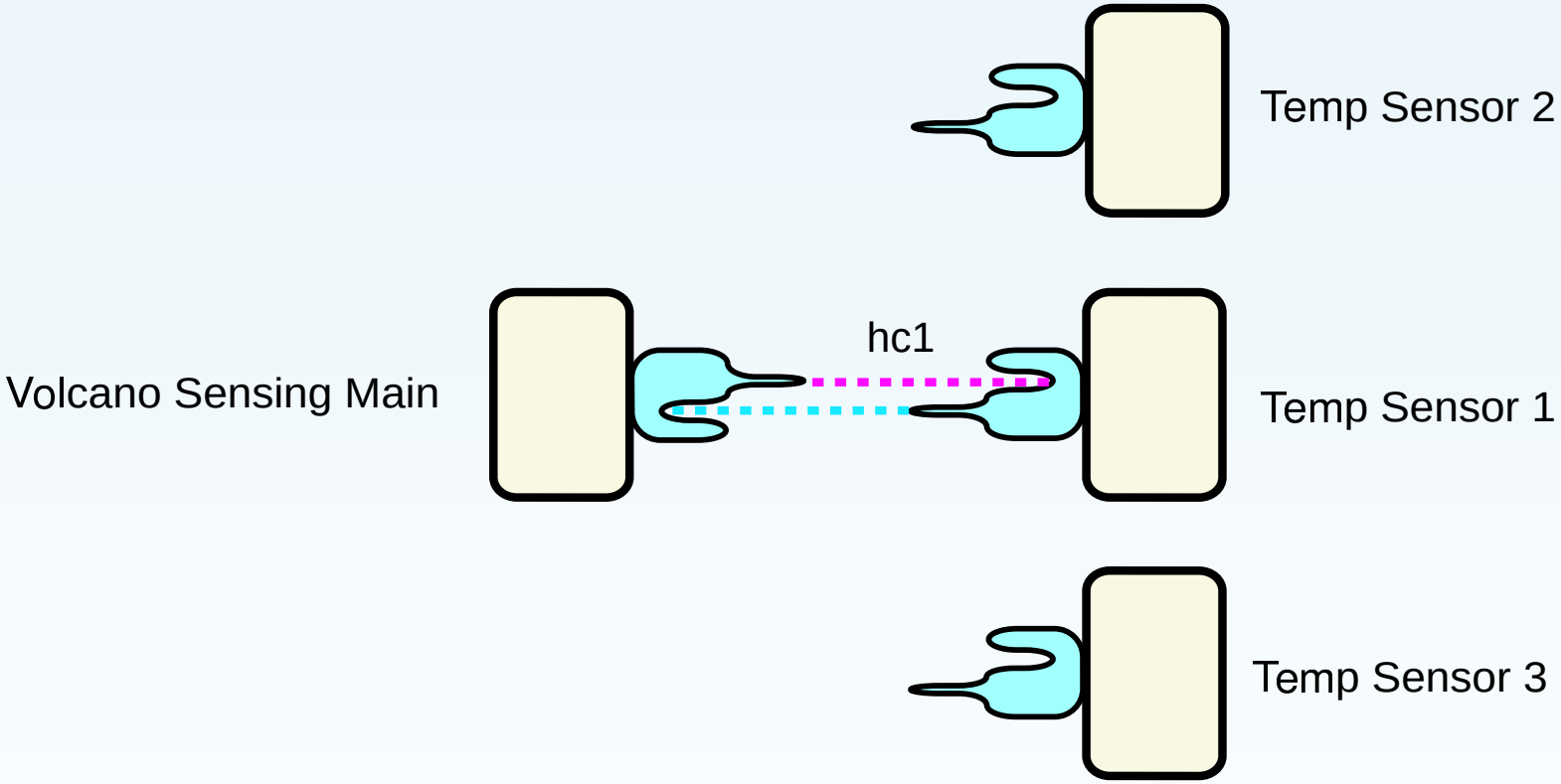
Distributed Messaging Illustrated



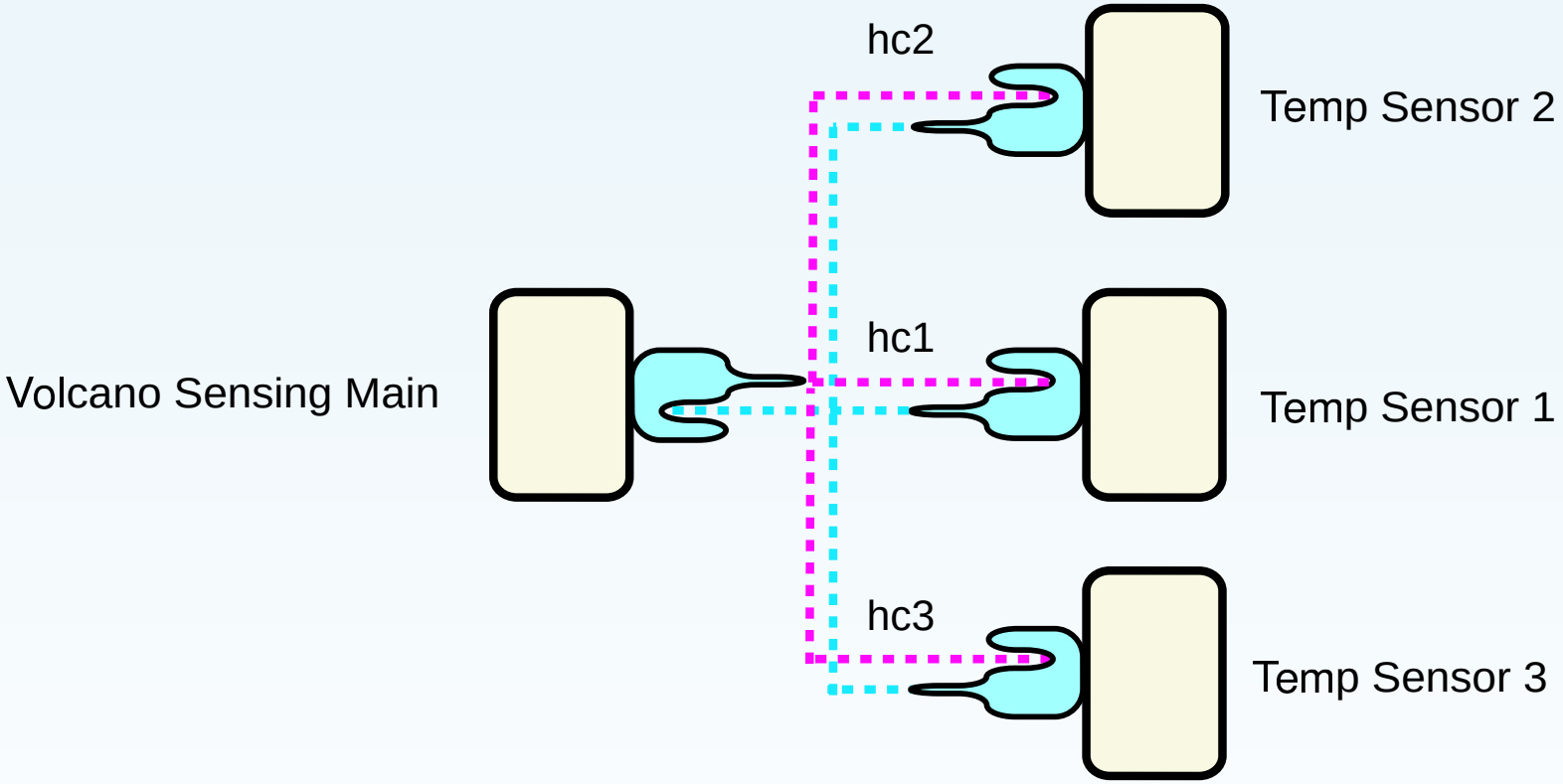
Distributed Messaging Illustrated



Distributed Messaging Illustrated



Distributed Messaging Illustrated



Data Encapsulation

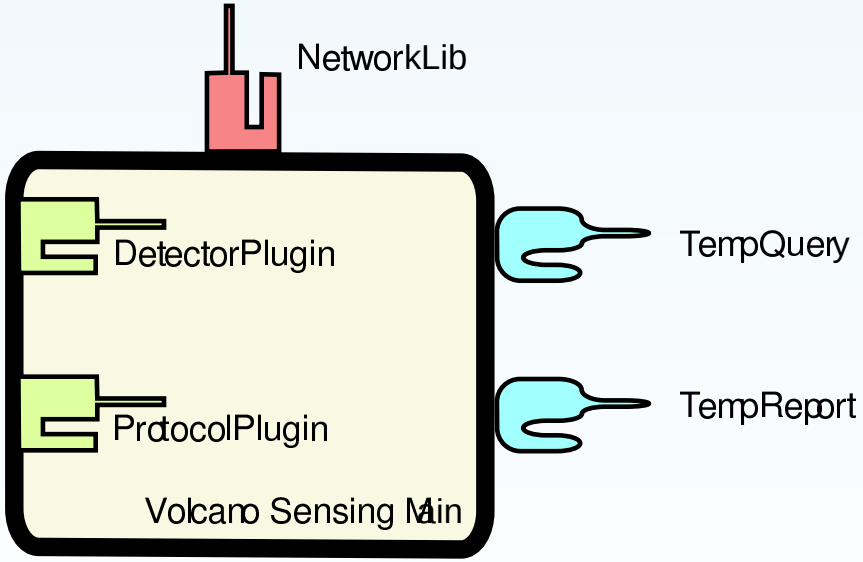
Goal: maintain sanctity of the runtime interfaces

- Only access internals of a runtime through its interfaces
- RMI/RPC violate the sanctity of interfaces
  - Object references can be passed around
  - No external interface on modules for potential RMI interactions

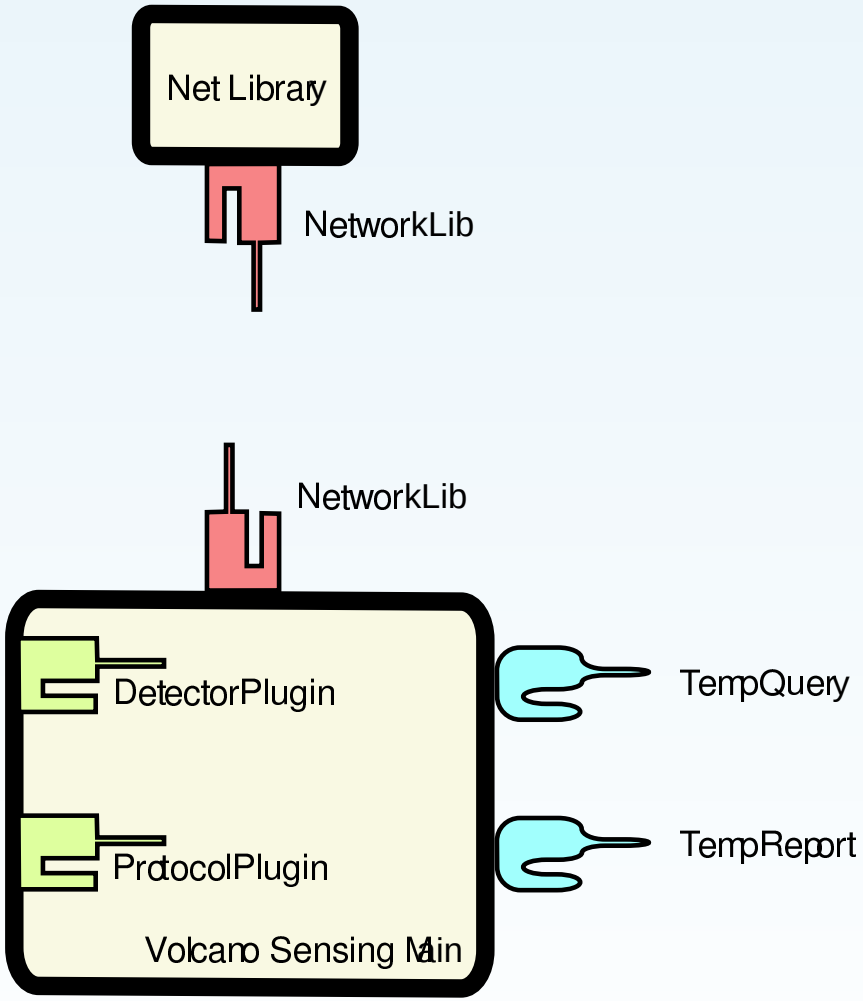
Thus we enforce the following

- No passing of object references across the network
- Objects can be copied, but *only* if their code is already plugged in on the other side
- Runtime references can be passed

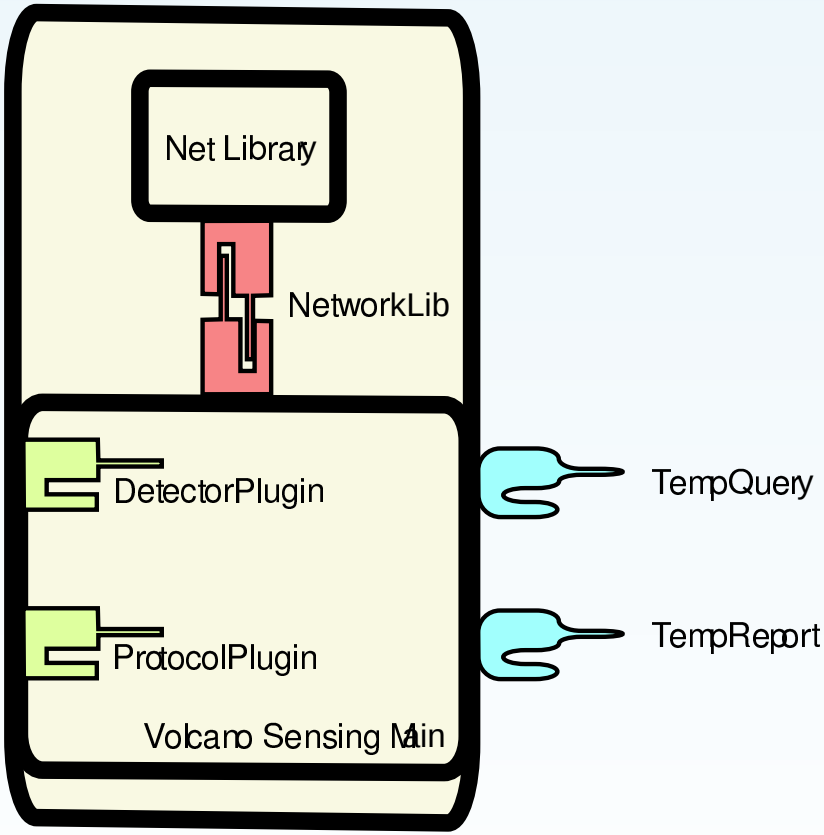
The Unification



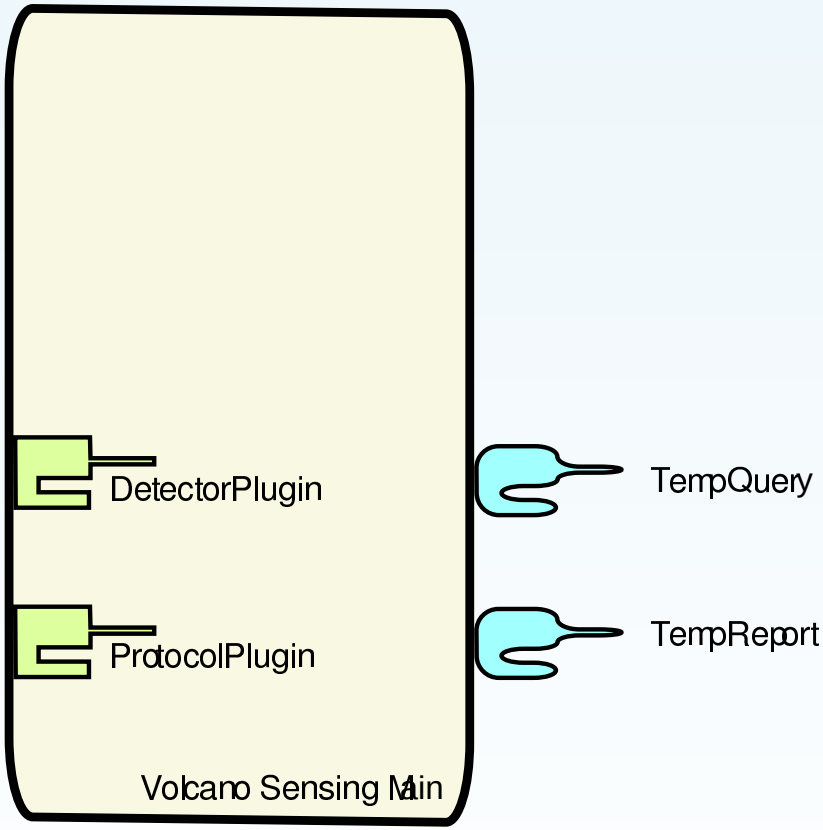
The Unification



The Unification



The Unification



The Code

VolcanoMain $=$ // Volcano sensor example

assemblage {

static linker NetLib {... statically linking some network library ...}

dynamic linker DetectorPlugin {

import detectMethod()

export getEnv $==$ ... get current environment snapshot ...}

connector CodeUpdate {

import getDetectCode();

export check($condition$) $==$ ... check applicability of detect model ...}

... // local feature implementation

updateDetector $== \lambda x.($ **let** $cb =$ **connect**$_{\text{CodeUpdate}\mapsto\text{Code}}$ x **in**

let $code = cb \triangleright$ getDetectCode $\leftarrow ()$ **in**

let $comp =$ **plugin**$_{\text{DetectorPlugin}\mapsto\text{Detect}}$ $code$ **in**

$comp$.detectMethod()))

...}

The Type System

- Static linkers, dynamic linkers, and connectors are typed
- Each import/export is typed
- There may be more exports than imports (subtyping)
- Assemblage runtime types are the connector types
- First-class assemblage types are static/dynamic linkers and connector types

Technical Accomplishments

- Formal operational semantics
- Type system
- Soundness via a subject reduction argument
  - Significant extra complication here due to multiple runtimes, modules as first-class values etc
- Extension to have types as features on linkers (not proved sound)

Related Work

Explicit interfaces are there to some degree

- Static linking
  - “solved” in many module systems/calculus.
- Dynamic linking
  - Java classloaders: not explicit; very fine-grained
  - Units, Argus, MJ, *etc* have partial interface support on code being linked, but not on the runtime
- Distributed langauge-based messaging
  - Existing protocols bury interactions in the code
  - RPC/RMI: When remote objects are passed between JVM’s, you don’t even know who you are talking to
  - Connectors

Future Work

- Implementation
- Ensemble: A connector-based Sensornet language
- MVM: a Microkernel Virtual Machine
- Extensions
  - Inference of signatures from assemblages
  - Linker- and connector-based security
  - Version control