Broker

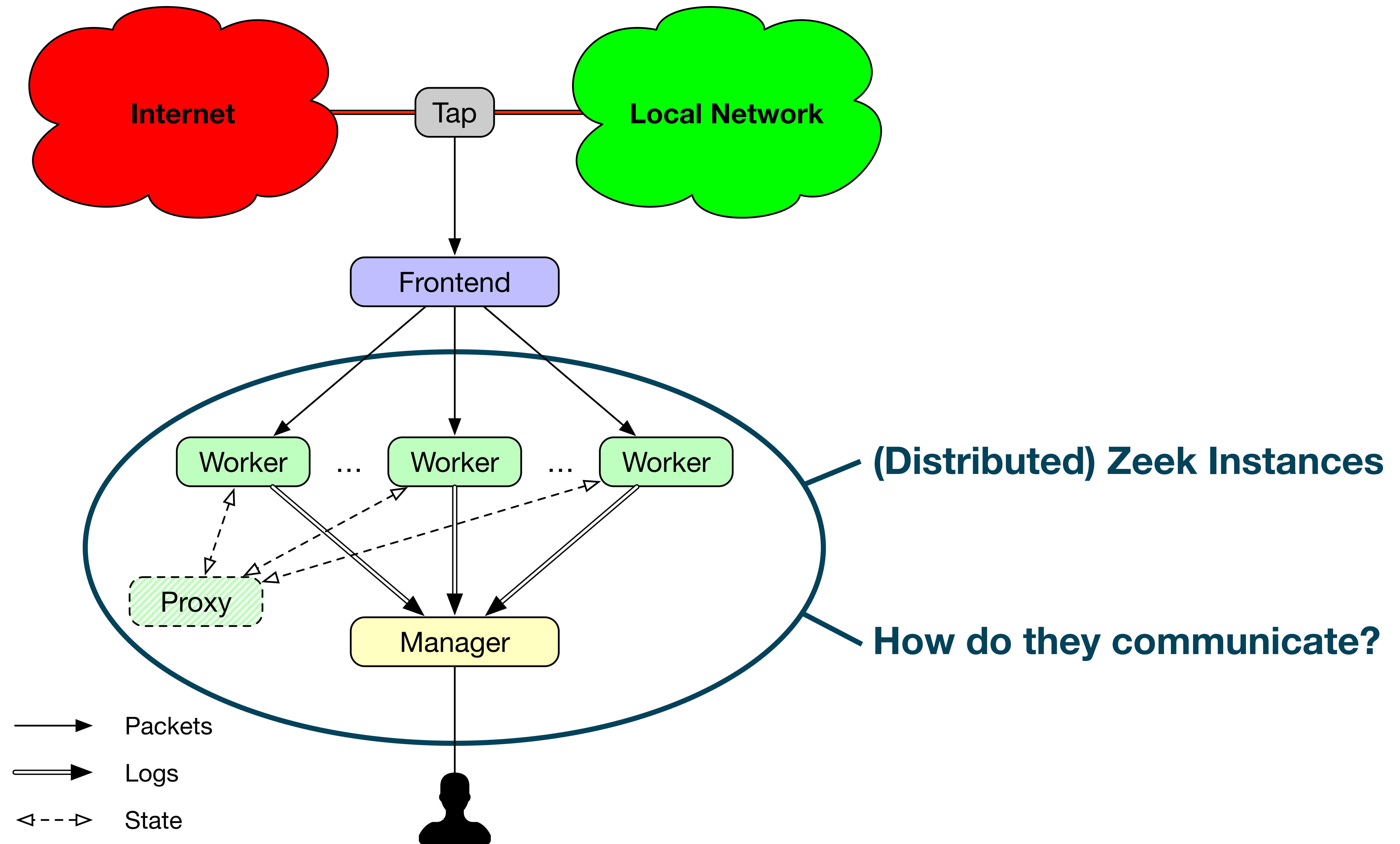
Zeek's Messaging Library

Dominik Charousset
http://dominik.charousset.de

May 2021

Zeek Clusters

An Idealized Zeek Cluster



Zeek Cluster Roles

- **Worker:** sniffs network traffic and runs protocol analysis
- **Manager:** collects events and creates a single, global view
- **Proxy:** offloads data storage or runs arbitrary work loads
- **Logger:** (optional) collects logs to reduce load on the manager

Zeek Cluster Challenges

- **Data dependencies:** which data an instance receives depends on the role
- **Flexible deployment:** users may add a logger and multiple proxies
- **State synchronization:** some modules / scripts need global view of events
- **Interfacing with 3rd parties:** users may want to integrate external tools

A Messaging Layer for Zeek

- We can fulfill our requirements with two building blocks:

  1. A topic-based **publish/subscribe layer**
     - Naturally models data dependencies via topics
     - Supports flexible deployments (publisher/subscriber rendezvous)
  2. Distributed **key/value stores**
     - Global lookup and updates of values for synchronization

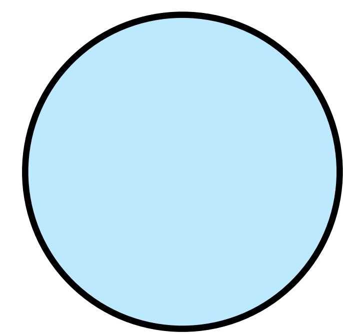
- Based on these considerations, implementing Broker started mid 2014

Broker Setup & API

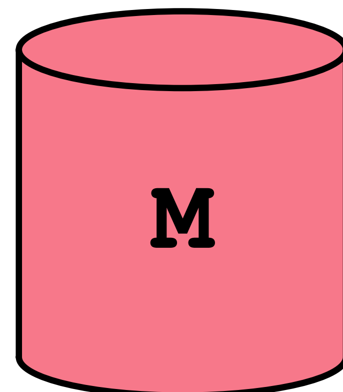
Overview

- Open Source C++ library with Python bindings (BSD-licensed)
- Available on GitHub: https://github.com/zeek/broker
- Requires recent versions of CMake, OpenSSL and CAF
- Usually comes bundled with Zeek but also works as standalone library

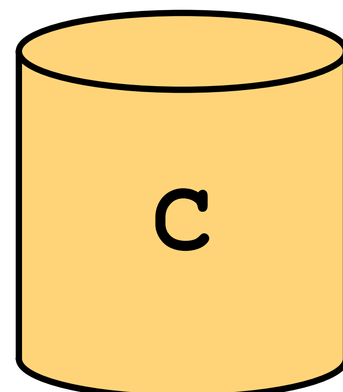
Terminology



A single Broker context / process

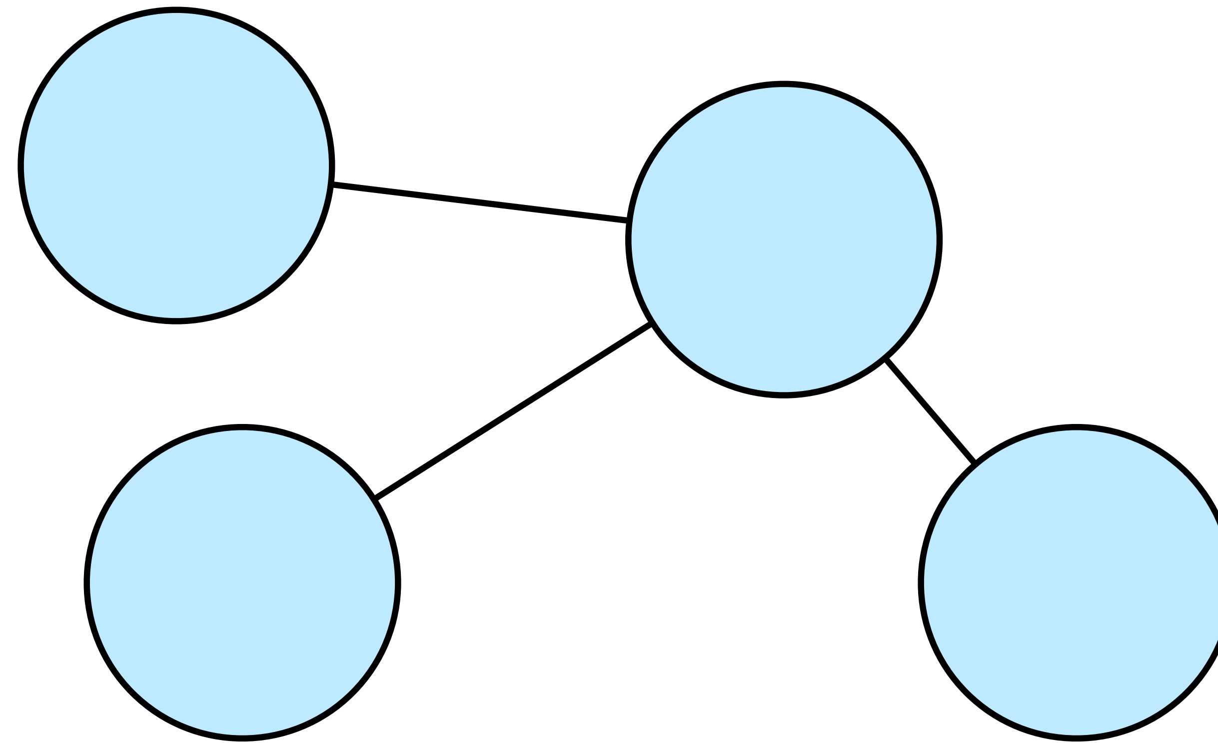


Authoritative data store source



Local data store cache

Endpoints



- Broker connects endpoints via *peering relations*
  - Each peering is a (TCP) network connection
  - Endpoints forward published data to peers with matching subscriptions

Peerings

- Require manual setup of the topology (no auto connections / discovery)
- To open a TCP port for incoming peerings: `listen(addr, port)`
- To connect to another endpoint: `peer(host, port)`
- **Caution**: Broker assumes loop-free topologies!
  - More on that later

Peering Setup: Zeek Scripts

```
redef exit_only_after_terminate = T;

event zeek_init()
    {
    Broker::listen("127.0.0.1");
    }

event Broker::peer_added(endpoint: Broker::EndpointInfo, msg: string)
    {
    print "peer added", endpoint;
    }

event Broker::peer_lost(endpoint: Broker::EndpointInfo, msg: string)
    {
    print "peer lost", endpoint;
    terminate();
    }
```

Passing no port uses `Broker::default_port` (9999 by default, but you can override the default via redef)

Listener

Peering Setup: Zeek Scripts

```
redef exit_only_after_terminate = T;

event zeek_init()
    {
    Broker::peer("127.0.0.1");
    }

event Broker::peer_added(endpoint: Broker::EndpointInfo, msg: string)
    {
    print "peer added", endpoint;
    terminate();
    }
```

Connector

Peering Setup: Python

```python
import broker

with broker.Endpoint() as ep:
    with ep.make_status_subscriber(True) as ssub:
        ep.listen("127.0.0.1", 9999)
        state = ssub.get()
        # On a successful connect, we see:
        # state.code() == broker.SC.PeerAdded
```

Listener

```python
import broker

with broker.Endpoint() as ep:
    with ep.make_status_subscriber(True) as ssub:
        ep.peer("127.0.0.1", 9999)
        state = ssub.get()
        # On a successful connect, we see:
        # state.code() == broker.SC.PeerAdded
```

Connector

Peering Setup: C++

```
 1 int main(int argc, char** argv) {
 2   using namespace broker;
 3   configuration cfg;
 4   cfg.init(argc, argv); // may throw!
 5   endpoint ep{std::move(cfg)};
 6   auto ssub = ep.make_status_subscriber(true);
 7   auto actual_port = ep.listen("127.0.0.1", 9999);
 8   if (actual_port == 0) {
 9     std::cerr << "unable to open port 9999\n";
10     return EXIT_FAILURE;
11   }
12   auto stat = ssub.get();
13   if (is<status>(stat)) {
14     // contains a status, e.g., sc::peer_added
15   } else if (is<error>(stat)) {
16     // contains an error, e.g., ec::peer_lost
17   }
18   return EXIT_SUCCESS;
19 }
```

Replace with `ep.peer(...)` for the connector.

Listener

Topics & Subscriptions

- Topics are encoded as (ASCII) strings, e.g., `foo/bar`
- Subscriptions match topics based on prefixes:
  - Subscribing to `foo/` matches `foo/bar`, but not `bar/foo` or `foobar`
  - Zeek & Broker use slash-delimited hierarchies by convention

Pub/Sub: Zeek Scripts Basics

```
redef exit_only_after_terminate = T;

global my_event: event(msg: string, c: count);
```

Subscriber

```
event zeek_init()
    {
    Broker::subscribe("zeek/event/");
    Broker::listen("127.0.0.1");
    }

event my_event(msg: string, c: count)
    {
    print "got my_event", msg, c;
    }
```

Publisher

```
event zeek_init()
    {
    Broker::peer("127.0.0.1");
    }

event Broker::peer_added(ep: Broker::EndpointInfo,
                         msg: string)
    {
    Broker::publish("zeek/event/my_event",
                    my_event, "hi", 0);
    }

event my_event(msg: string, c: count)
    {
    print "got my_event", msg, c;
    }
```

Triggers `my_event` handlers on both sides!

Pub/Sub: Zeek Scripts Magic

```
redef exit_only_after_terminate = T;

global my_event: event(msg: string, c: count);
```

Subscriber

```
event zeek_init()
    {
    Broker::subscribe("zeek/event/");
    Broker::listen("127.0.0.1");
    }

event my_event(msg: string, c: count)
    {
    print "got my_event", msg, c;
    }
```

Publisher

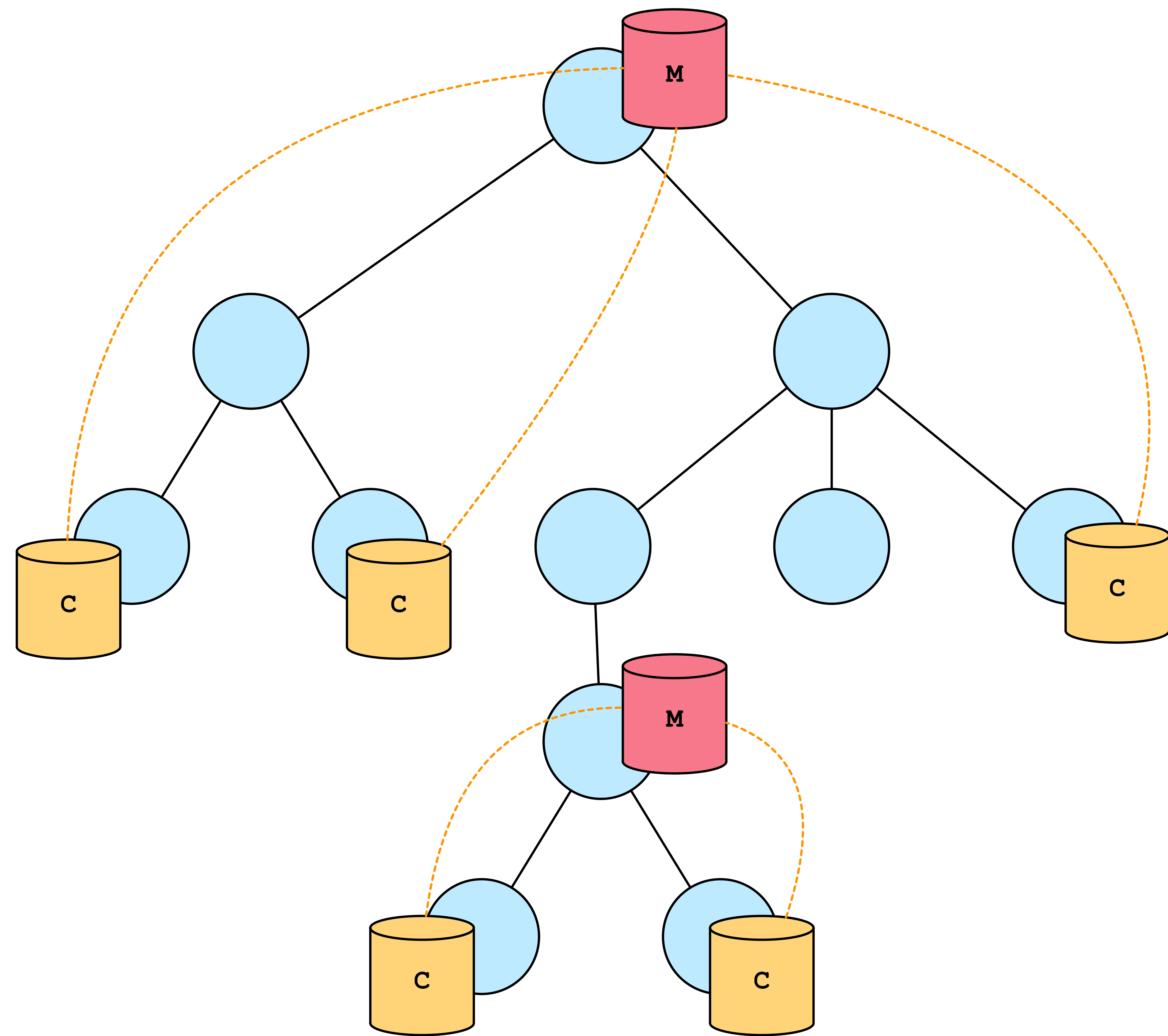
```
event zeek_init()
    {
    Broker::peer("127.0.0.1");
    Broker::auto_publish("zeek/event/my_event",
                         my_event);
    }

event Broker::peer_added(ep: Broker::EndpointInfo,
                         msg: string)
    {
    event my_event("hi", 0);
    }

event my_event(msg: string, c: count)
    {
    print "got my_event", msg, c;
    }
```

Triggers `my_event` handlers on both sides via implicit call to `Broker::publish`.

Pub/Sub Summary

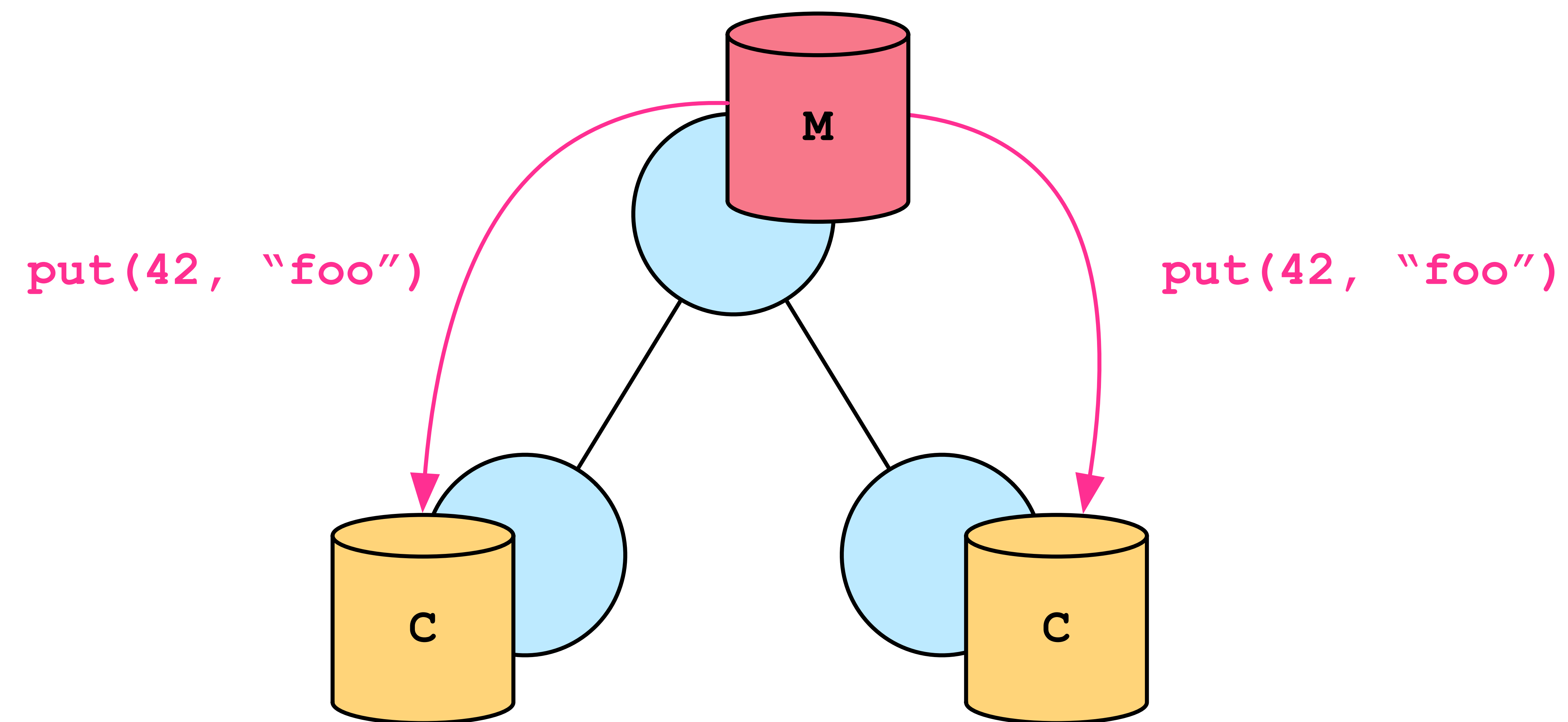
- Zeek maps Broker messages to events
- General advise: subscribe before peer
  - New subscriptions need some time to propagate
  - Published data cannot be “re-captured” later (no buffering)
- Python and C++: `publish` and `subscribe` functions (blocking & async)

Data Stores



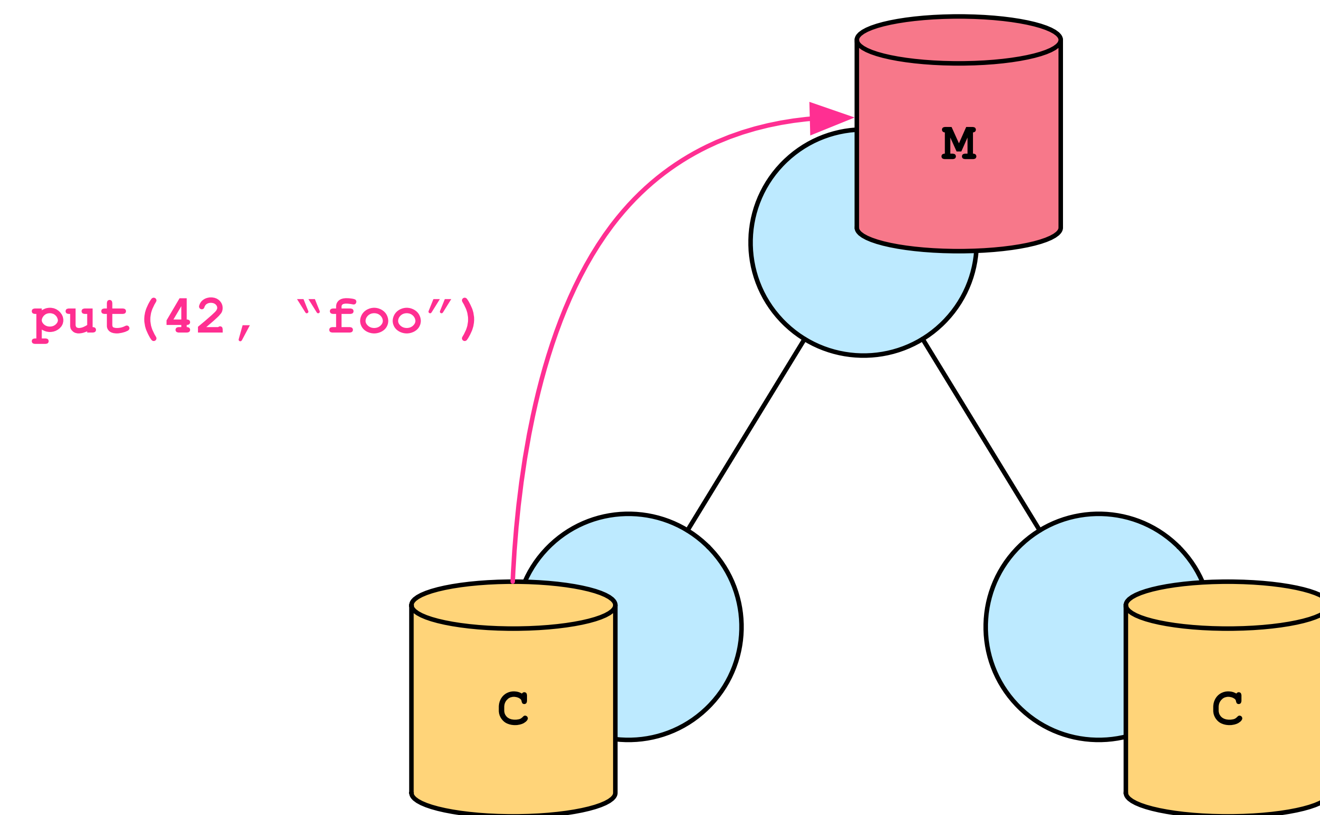
- Masters & clones attach to endpoints
- “Double duty” for peerings:
  - Pub/Sub traffic
  - Data store commands

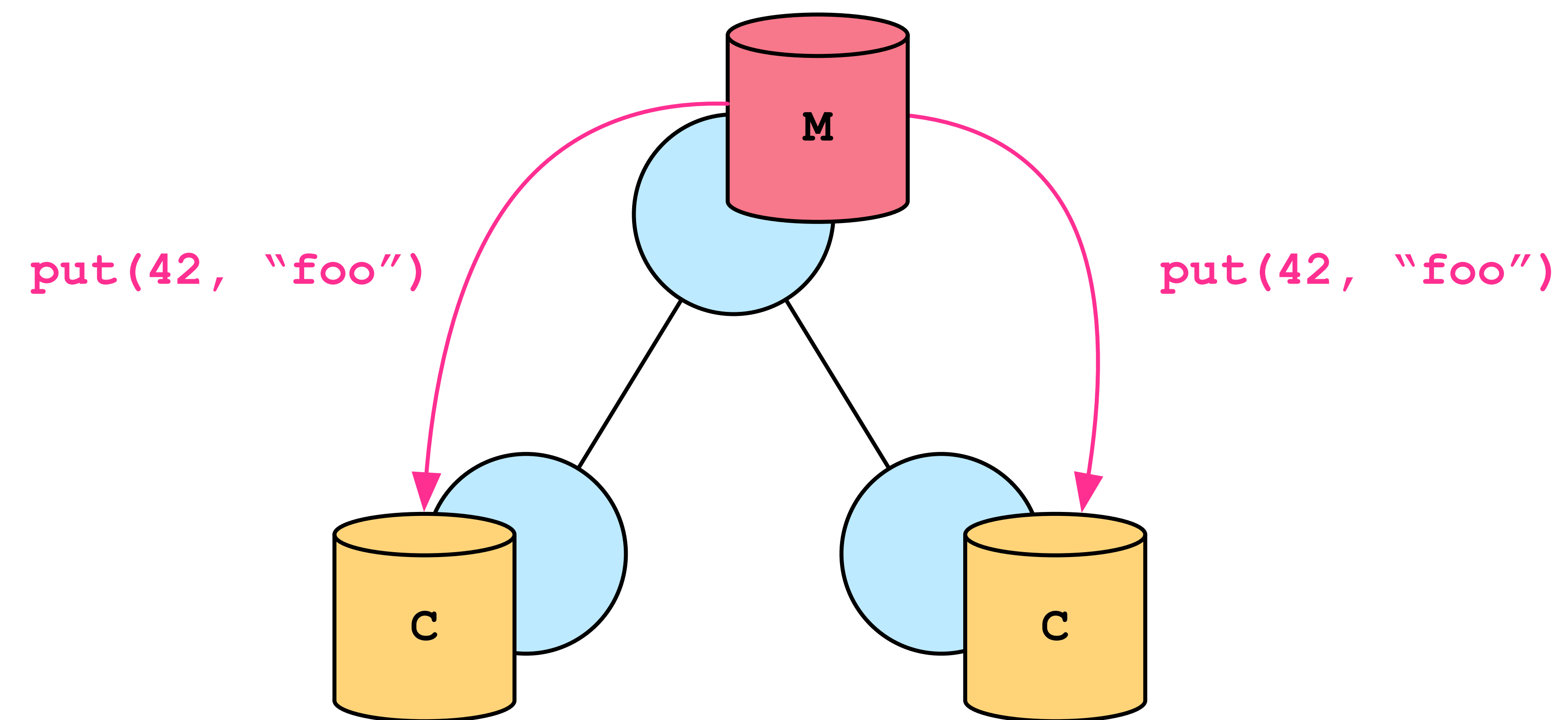
Date Store Writes



Date Store Writes

Modification through clone:
centralized replay through master





1 send operation to master

2 apply and replay operation

Data Stores in Zeek Scripts

```
global h: opaque of Broker::Store;

event zeek_init()
    {
    h = Broker::create_master("mystore");
    # or: h = Broker::create_clone("mystore");

    # writing
    Broker::put(h, "one", 110);
    Broker::increment(h, "one");
    local myset: set[string] = {"a", "b", "c"};
    Broker::put(h, "myset", myset);
    Broker::insert_into_set(h, "myset", "d");
    Broker::remove_from(h, "myset", "b");

    # reading
    local res = Broker::get(h, "one")
    print "one: ", res;
    }
```

Date Store Features

- Increment/decrement operations for atomic updates on numbers
- Add/remove functions for atomic updates on sets etc.
- Key-value pairs optionally have an expiration time
- Zeek can automagically synchronize table contents across clusters:
  - `global t: table[string] of count &backend=Broker::MEMORY;`
- Broker includes an SQLite backend for persistent state

Limitations & Outlook

Current Limitations

- Broker assumes loop-free topologies
  - Simplifies forwarding logic and requires little state
  - But: easy to misconfigure and no “fallback” routes on link errors
- Rigid peering connection hinder more use cases

Broker in Zeek Clusters Today

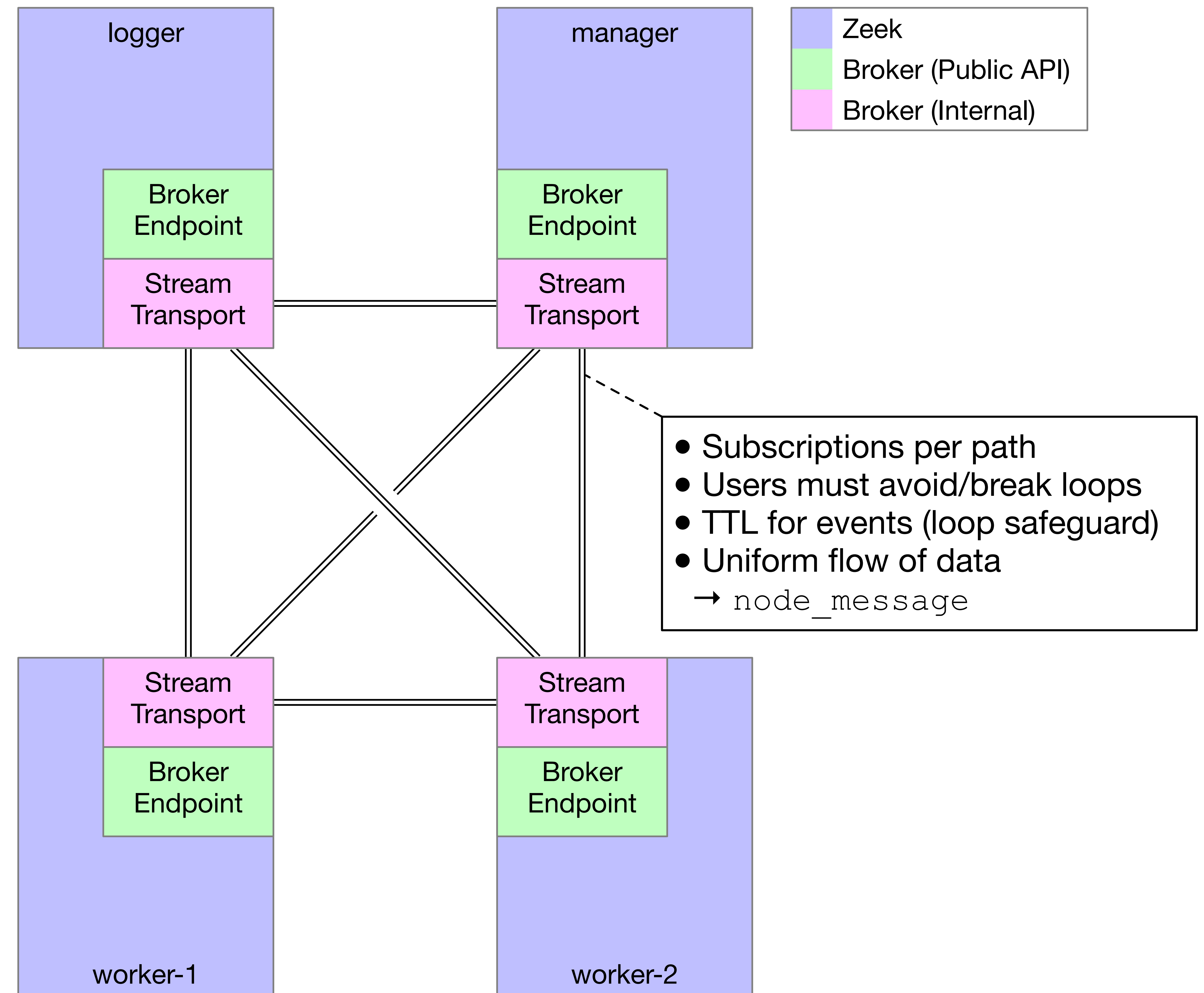
- Based on simple TCP steam sockets
- Endpoints see only direct peers / connections
- State (subscriptions, forwarding flags) remains mostly on the paths

Pro

- Little state per node
- Simple dispatching logic

Conn

- Easy to misconfigure
- No redundancies
- Topology opaque



Introducing ALM

- Goal: enable more use cases for Broker and increase robustness
- To overcome current restrictions, we combine:
  - Application Layer Multicast (ALM) to express pub/sub on a higher level
  - Source Routing to safely operate on “loopy” topologies

Next-Gen Broker with Zeek

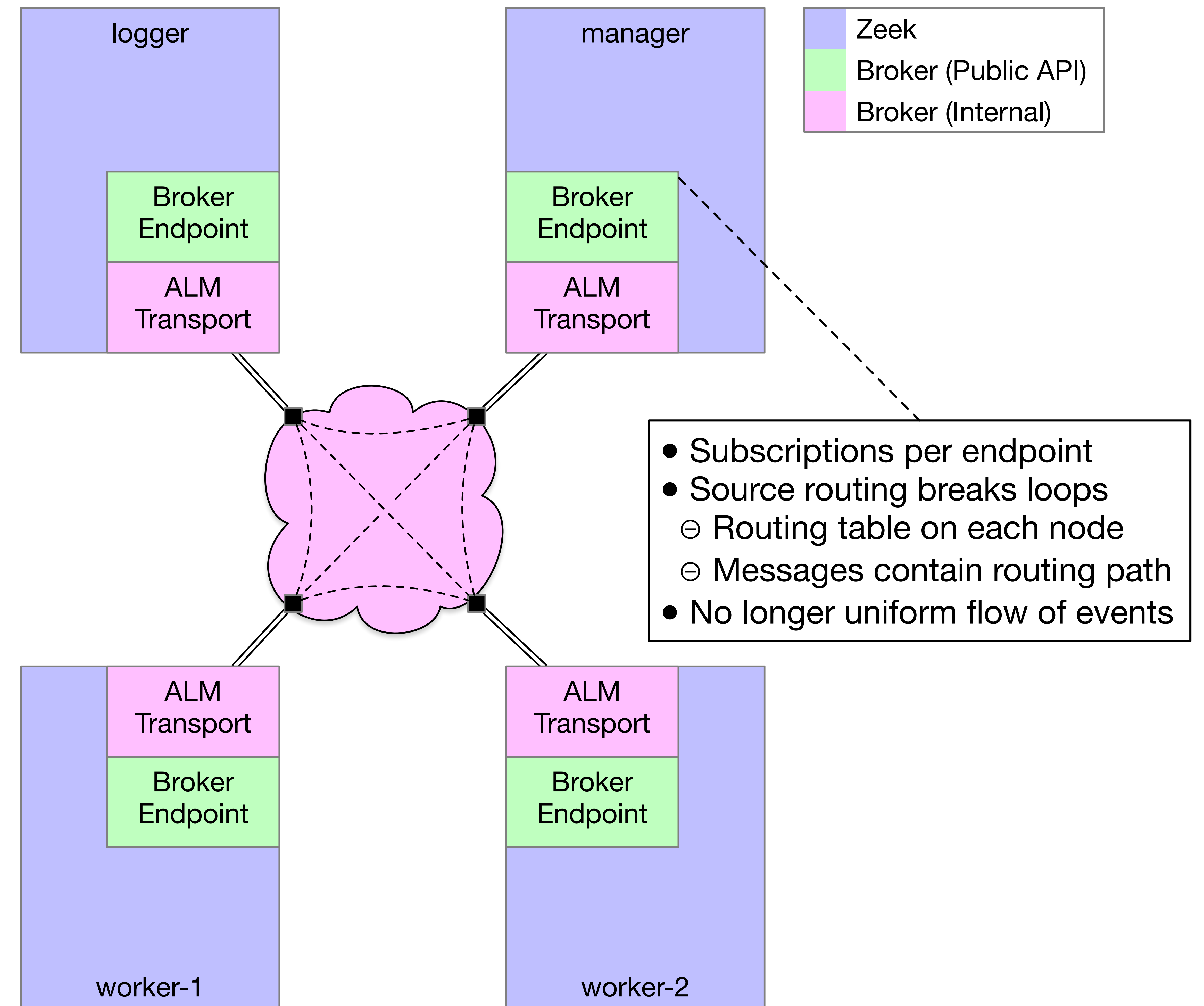
- Based on Peer-to-Peer (P2P) Networking
- Full visibility of cluster topology (exception: Gateways)
- State (subscriptions, routing) on the endpoints

Pro

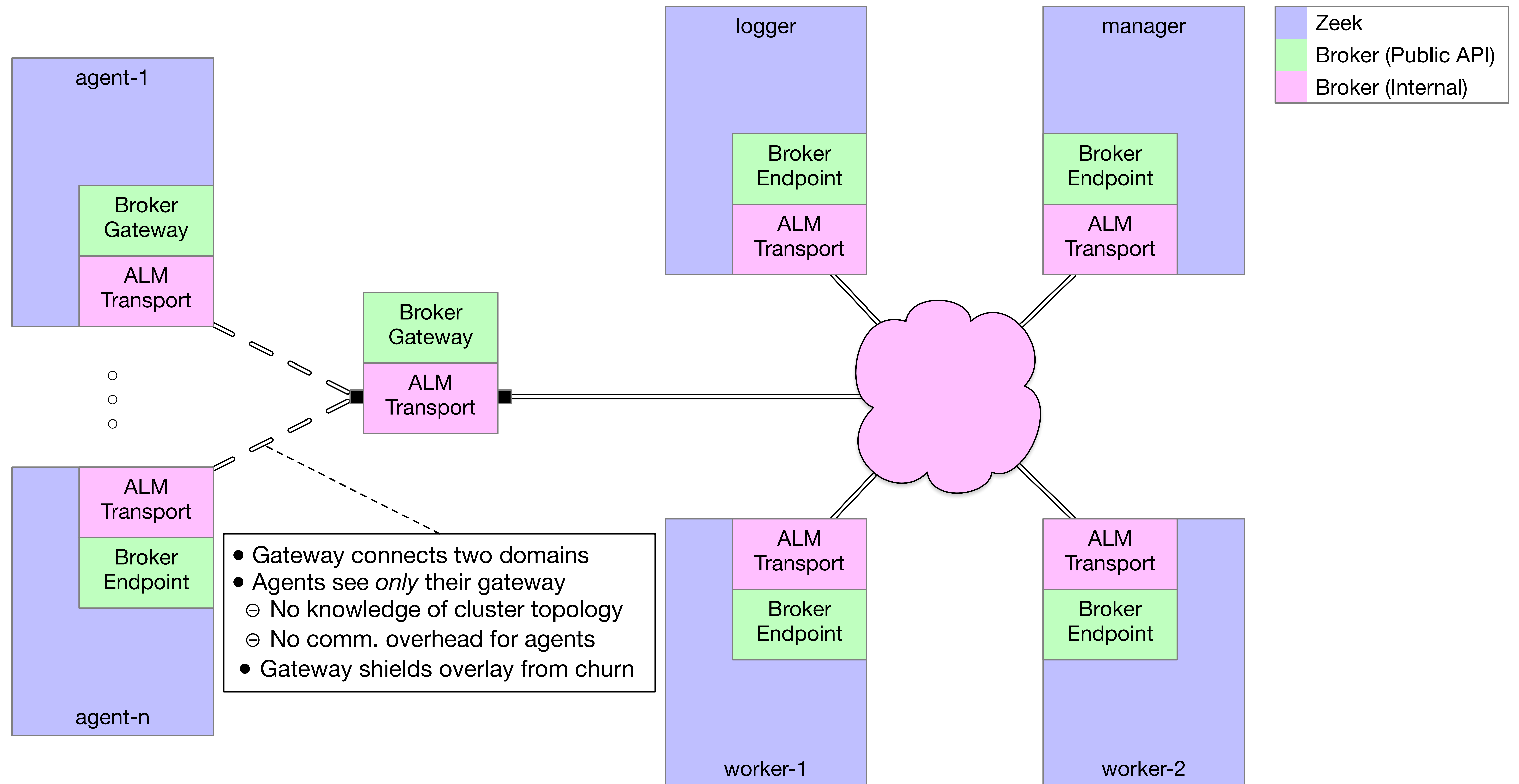
- Topology well known
- Forwarding *just works*
- Loops *add* resilience
- Enables new use cases, e.g., connecting Zeek Agents

Conn

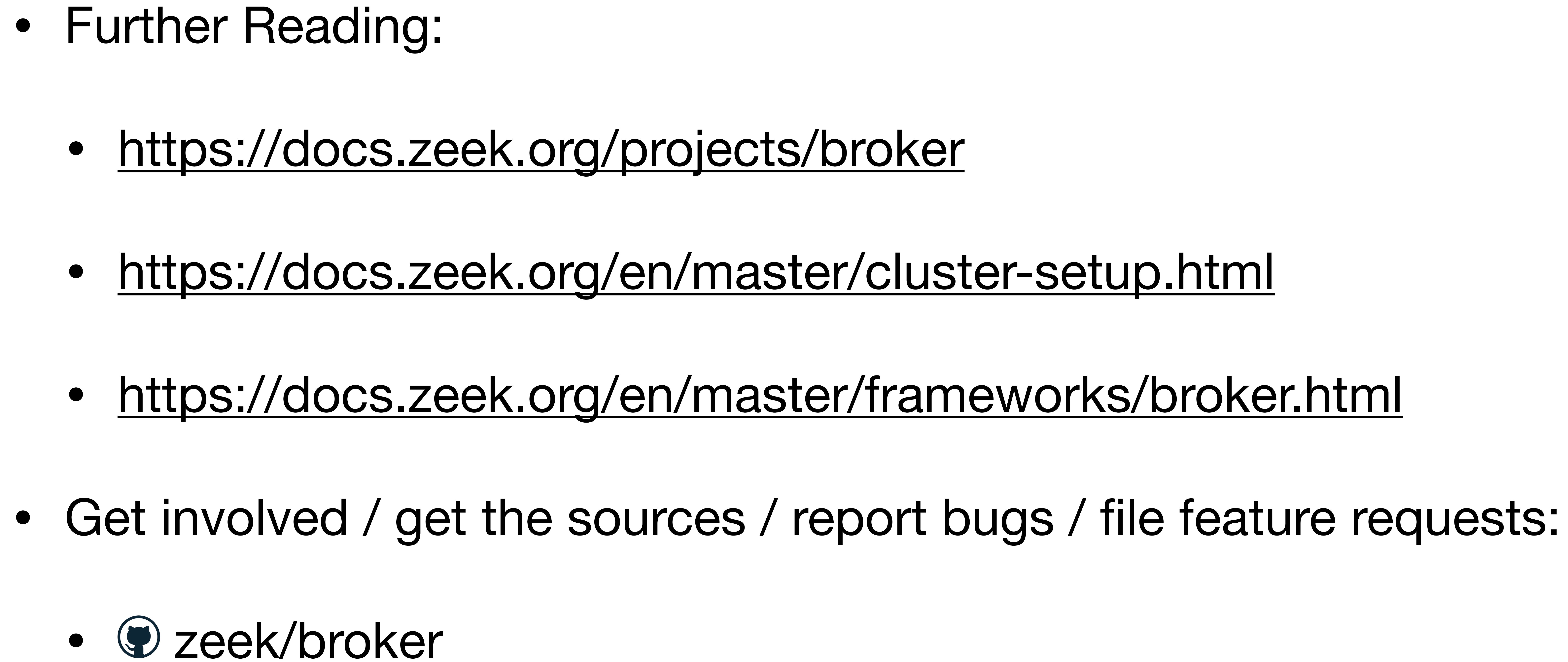
- More state per node
- More traffic
  - Subscr. flooding
  - Routing headers



Connecting Zeek Agents



Thank You for Joining Today!



- Further Reading:
  - https://docs.zeek.org/projects/broker
  - https://docs.zeek.org/en/master/cluster-setup.html
  - https://docs.zeek.org/en/master/frameworks/broker.html
- Get involved / get the sources / report bugs / file feature requests:
  - zeek/broker