Broker

Release 1.2.8

Apr 30, 2020

Contents

Broker is a library for type-rich publish/subscribe communication in Zeek's data model.

CHAPTER 1

Outline

Section 1 introduces Broker's key components and basic terminology, such as *endpoints*, *messages*, *topics*, and *data stores*.

Section 2 shows how one can send and receive data with Broker's publish/subscribe communication primitives. By structuring applications in independent *endpoints* and peering with other endpoints, one can create a variety of different communication topologies that perform topic-based message routing.

Section 3 presents Broker's data model, which applications can pack into messages and publish under given topics. The same data model is also used by Broker's *data stores*.

Section 4 introduces *data stores*, a distributed key-value abstraction operating with the complete *data model*, for both keys and values. Users interact with a data store *frontend*, which is either an authoritative *master* or a *clone* replica. The master can choose to keep its data in various *backends*, currently either in-memory, or persistently through SQLite, or RocksDB.

Section 5 discusses the Broker's Python bindings, which transparently expose all of the library's functionality to Python scripts.

CHAPTER 2

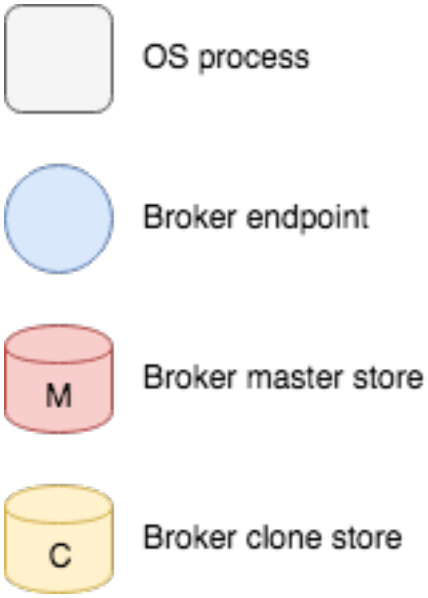
Synopsis

```cpp
#include <iostream>
#include <broker/broker.hh>

using namespace broker;

int main()
{
    endpoint ep;
    ep.peer("1.2.3.4", 9999); // Connect to remote endpoint on given address/port.

    // Messages

    ep.publish("/test/t1", set{1, 2, 3}); // Publish data under a given topic.

    auto sub = ep.make_subscriber({"/test/t2"}); // Subscribe to incoming messages␣
↪for topic.
    auto msg = sub.get(); // Wait for one incoming message.
    std::cout << "got data for topic " << get_topic(msg) << ": " << get_data(msg) <<␣
↪std::endl;

    // Data stores

    auto m = ep.attach_master("yoda", backend::memory); // Create data store.

    m->put(4.2, -42); // Write into store.
    m->put("bar", vector{true, 7u, now()});

    if ( auto d = m->get(4.2) ) // Look up in store.
        std::cout << "value of 4.2 is " << d << std::endl;
    else
        std::cout << "no such key: 4.2" << std::endl;
}
```

2.1 Overview

The **Broker** library enables applications to communicate in Zeek's type-rich *data model* via publish/subscribe messaging. Moreover, Broker offers distributed *key-value stores* to facilitate unified data management and persistence.

The figure below introduces the graphic terminology we use throughout this manual.

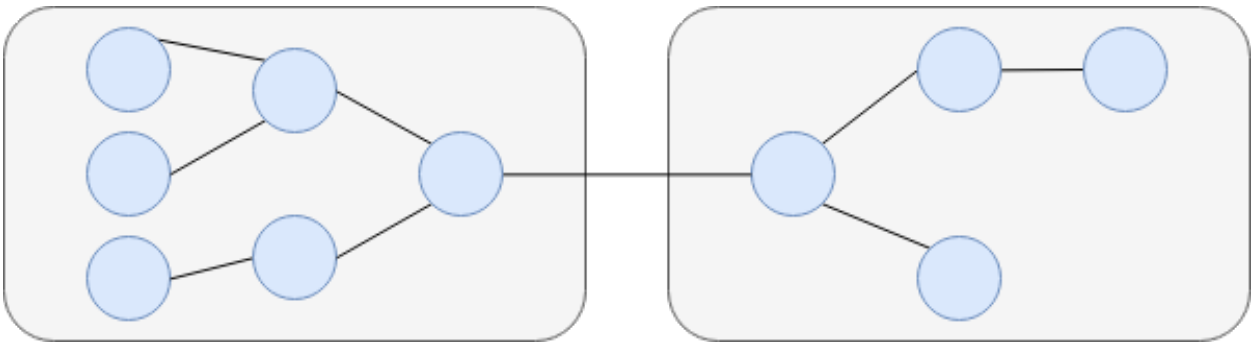


Moreover, all C++ code examples assume `using namespace broker` for conciseness.

2.1.1 Communication

Broker structures an application in terms of *endpoints*, which represent data senders and receivers. Endpoints can peer with other endpoints to communicate with their neighbors. An endpoint can send a message to its peers by publishing data under a specific *topic*. If any endpoint holds a subscription to the topic, it will receive the corresponding data.

Endpoints can efficiently communicate within the same OS process, as well as transparently communicate with endpoints in a different OS process or on a remote machine. For in-memory endpoints, sending a message boils down to passing a pointer. For remote communication, Broker serializes messages transparently. This allows for a variety of different communication patterns. The following figure illustrates an exemplary topology.



A process hosts one or more endpoints. Endpoints can communicate within or across processes as well as machine boundaries.

The fundamental unit of exchange is a *message*, which consists of a *topic* and *data*. Endpoints may choose to forward received messages to their own peers that share a matching topic.

The API allows for both synchronous and asynchronous communication. Internally, Broker operates entirely asynchronously by leveraging the C++ Actor Framework (CAF). Users can receive messages either explicitly polling for them, or by installing a callback to execute as they come in.

See *Section 2* for concrete usage examples.

2.1.2 Data Model

Broker comes with a rich data model, since the library's primary objective involves communication with Zeek and related applications. The fundamental unit of communication is `data`, which can hold any of the following concrete types:

- `none`
- `boolean`
- `count`
- `integer`
- `real`
- `timespan`
- `timestamp`
- `string`
- `address`
- `subnet`
- `port`
- `vector`
- `set`
- `table`

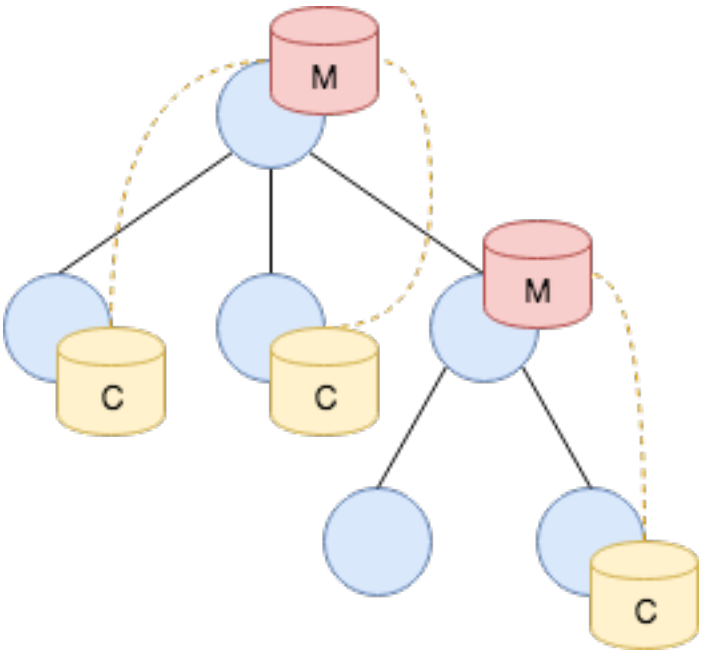
Section 3 discusses the various types and their API in depth.

From these data units, one then composes *messages* to be exchanged. Broker does generally not impose any further structure on messages, it's up to sender and receiver to agree. For communication with Zeek, however, Broker provides an additional *event* abstraction that defines the specific message layout that Zeek expects for exchanging Zeek events.

2.1.3 Data Stores

Data stores complement endpoint communication with a distributed key-value abstraction operating in the full data model. One can attach one or more data stores to an endpoint. A data store has a *frontend*, which determines its behavior, and a *backend*, which represents the type of database for storing data. There exist two types of frontends: *master* and *clone*. A master is the authoritative source for the key-value store, whereas a clone represents a local cache. Only the master can perform mutating operations on the store, which it then pushes to all its clones over the existing peering communication channel. A clone has a full copy of the data for faster access, but transparently sends any modifying operations to its master first. Only when the master propagates back the change, the result of the operation becomes visible at the clone. The figure below illustrates how one can deploy a master with several clones.

Each data store has a name that identifies the master. This name must be unique among the endpoint's peers. The master can choose to keep its data in various backends, which are currently: in-memory, SQLite, and RocksDB.



Section 4 illustrates how to use data stores in different settings.

2.2 Communication

Broker's primary objective is to facilitate efficient communication through a publish/subscribe model. In this model, entities send data by publishing to a specific topic, and receive data by subscribing to topics of interest. The asynchronous nature of publish/subscribe makes it a popular choice for loosely coupled, distributed systems.

Broker is the successor of Broccoli. Broker enables arbitrary applications to communicate in Zeek's data model. In this chapter, we first describe generic Broker communication between peers that don't assume any specific message layout. Afterwards, we show how to exchange events with Zeek through an additional Zeek-specific shim on top of Broker's generic messages.

2.2.1 Exchanging Broker Messages

We start with a discussion of generic message exchange between Broker clients. At the Broker level, messages are just arbitrary values that have no further semantics attached. It's up to senders and receivers to agree on a specific layout of messages (e.g., a set of doubles for a measurement series).

Endpoints

Broker encapsulates its entire peering setup in an `endpoint` object. Multiple instances of an `endpoint` can exist in the same process, but each `endpoint` features a thread-pool and (configurable) scheduler, which determines the execution of Broker's components. Using a single `endpoint` per OS process guarantees the most efficient usage of available hardware resources. Nonetheless, multiple Broker applications can seamlessly operate when linked together, as there exists no global library state.

Note: Instances of type `endpoint` have reference semantics: that is, they behave like a reference in that it's impossible to obtain an invalid one (unlike a null pointer). An `endpoint` can also be copied around cheaply, but is

not safe against access from concurrent threads.

Peerings

In order to publish or receive messages an endpoint needs to peer with other endpoints. A peering is a bidirectional relationship between two endpoints. Peering endpoints exchange subscriptions and then forward messages accordingly. This allows for creating flexible communication topologies that use topic-based message routing.

An endpoint can either initiate a peering itself by connecting to remote locations, or wait for an incoming request:

```
// Open port and subscribe to 'foo' with all
// incoming peerings.
// Establish outgoing peering and subscribe to 'bar'.
endpoint ep1;
auto sub1 = ep1.make_subscriber({"/topic/test"});
ep1.peer("127.0.0.1", 9999);

endpoint ep0;
auto sub0 = ep0.make_subscriber({"/topic/test"});
ep0.listen("127.0.0.1", 9999);
```

Sending Data

In Broker a message consists of a *topic-data* pair. That is, endpoints *publish* values as *data* instances along with a *topic* that steers them to interested subscribers:

```
ep.publish("/topic/test", "42"); // Message is a single number.
ep.publish("/topic/test", vector{1, 2, 3}); // Message is a vector of values.
```

Note: Publishing a message can be a no-op if there exists no subscriber. Because Broker has fire-and-forget messaging semantics, the runtime does not generate a notification if no subscribers exist.

One can also explicitly create a dedicated `publisher` for a specific topic first, and then use that to send subsequent messages. This approach is better suited for high-volume streams, as it leverages CAF's demand management internally:

```
auto pub = ep.make_publisher("/topic/test");
pub.publish("42"); // Message is a single number.
pub.publish(vector{1, 2, 3}); // Message is a vector.
```

Finally, there's also a streaming version of the publisher that pulls messages from a producer as capacity becomes available on the output channel; see `endpoint::publish_all` and `endpoint::publish_all_no_sync`.

See *Section 3* for a detailed discussion on how to construct values for messages in the form of various types of `data` instances.

Receiving Data

Endpoints receive data by creating a `subscriber` attached to the topics of interest. Subscriptions are prefix-based, matching all topics that start with a given string. A `subscriber` can either retrieve incoming messages explicitly by calling `get` or `poll` (synchronous API), or spawn a background worker to process messages as they come in (asynchronous API).

Synchronous API

The synchronous API exists for applications that want to poll for messages explicitly. Once a subscriber is registered for topics, calling `get` will wait for a new message:

```
endpoint ep;
auto sub = ep.make_subscriber({"/topic/test"});
auto msg = sub.get();
auto topic = get_topic(msg);
auto data_ = get_data(msg);
std::cout << "topic: " << topic << " data: " << data_ << std::endl;
```

By default the function `get` blocks until the subscriber has at least one message available, which it then returns. Each retrieved message consists of the same two elements that the publisher passed along: the topic that the message has been published to, and the message's payload in the form of an arbitray Broker value, (i.e., a *data* instance). The example just prints them both out.

Blocking indefinitely until messages arrive often won't work well, in particular not in combination with existing event loops or polling. Therefore, `get` takes an additional optional timeout parameter to wait only for a certain amount of time. Alternatively, one can also use `available` to explicitly check for available messages, or `poll` to extract just all currently pending messages (which may be none):

```
if ( sub.available() )
    msg = sub.get(); // Won't block now.

for ( auto m : sub.poll() ) // Iterate over all available messages
    std::cout << "topic: " << get_topic(m) << " data: " << get_data(m) << std::endl;
```

For integration into event loops, `subscriber` also provides a file descriptor that signals whether messages are available:

```
auto fd = sub.fd();
::pollfd p = {fd, POLLIN, 0};
auto n = ::poll(&p, 1, -1);
if (n < 0)
    std::terminate(); // poll failed

if (n == 1 && p.revents & POLLIN) {
    auto msg = sub.get(); // Won't block now.
    // ...
    }
```

Asynchronous API

TODO: Document.

Status and Error Messages

Broker informs clients about any communication errors—and optionally also about non-critical connectivity changes—through separate `status` messages. To get access to that information, one creates a `status_subscriber`, which provides a similar synchronous `get/available/poll` API as the standard message subscriber. By default, a `status_subscriber` returns only errors:

```
auto ss = ep.make_status_subscriber();

if ( ss.available() ) {
    auto ss_res = ss.get();
    auto err = caf::get<error>(ss_res); // Won't block now.
    std::cerr << "Broker error:" << err.code() << ", " << to_string(err) << std::endl;
}
```

Errors reflect failures that may impact the correctness of operation. `err.code()` returns an enum `ec` that codifies existing error codes:

```
/// @relates status
enum class ec : uint8_t {
  /// The unspecified default error code.
  unspecified = 1,
  /// Version incompatibility.
  peer_incompatible,
  /// Referenced peer does not exist.
  peer_invalid,
  /// Remote peer not listening.
  peer_unavailable,
  /// An peering request timed out.
  peer_timeout,
  /// Master with given name already exist.
  master_exists,
  /// Master with given name does not exist.
  no_such_master,
  /// The given data store key does not exist.
  no_such_key,
  /// The store operation timed out.
  request_timeout,
  /// The operation expected a different type than provided
  type_clash,
  /// The data value cannot be used to carry out the desired operation.
  invalid_data,
  /// The storage backend failed to execute the operation.
  backend_failure,
  /// The clone store has not yet synchronized with its master, or it has
  /// been disconnected for too long.
  stale_data,
```

To receive non-critical status messages as well, specify that when creating the `status_subscriber`:

```
auto ss = ep.make_status_subscriber(true); // Get status updates and errors.

if ( ss.available() ) {
    auto s = ss.get();

    if ( auto err = caf::get_if<error>(&s) )
        std::cerr << "Broker error:" << err->code() << ", " << to_string(*err) <<␣
↪std::endl;

    if ( auto st = caf::get_if<status>(&s) ) {
        if ( auto ctx = st->context<endpoint_info>() ) // Get the peer this is about␣
↪if available.
            std::cerr << "Broker status update regarding "
                      << ctx->network->address
```

(continues on next page)

(continued from previous page)

```
                    << ":" << to_string(*st) << std::endl;
        else
            std::cerr << "Broker status update:"
                    << to_string(*st) << std::endl;
    }
```

Status messages represent non-critical changes to the topology. For example, after a successful peering, both endpoints receive a `peer_added` status message. The concrete semantics of a status depend on its embedded code, which the enum `sc` codifies:

```
enum class sc : uint8_t {
  /// The unspecified default error code.
  unspecified = 0,
  /// Successfully added a new peer.
  peer_added,
  /// Successfully removed a peer.
  peer_removed,
  /// Lost connection to peer.
  peer_lost,
};
```

Status messages have an optional *context* and an optional descriptive *message*. The member function `context<T>` returns a `const T*` if the context is available. The type of available context information is dependent on the status code enum `sc`. For example, all `sc::peer_*` status codes include an `endpoint_info` context as well as a message.

2.2.2 Forwarding

In topologies where multiple endpoints are connected, an endpoint forwards incoming messages to peers by default for topics that it is itself subscribed to. One can configure additional topics to forward, independent of the local subscription status, through the method `endpoint::forward(std::vector<topics>)`. One can also disable forwarding of remote messages altogether through the Broker configuration option `forward` when creating an endpoint.

When forwarding messages Broker assumes all connected endpoints form a tree topology without any loops. Still, to avoid messages circling indefinitely if a loop happens accidentally, Broker's message forwarding adds a TTL value to messages, and drops any that have traversed that many hops. The default TTL is 20; it can be changed by setting the Broker configuration option `ttl`. Note that it is the first hop's TTL configuration that determines a message's lifetime (not the original sender's).

2.2.3 Exchanging Zeek Events

The communication model discussed so far remains generic for all Broker clients in that it doesn't associate any semantics with the values exchanged through messages. In practice, however, senders and receivers will need to agree on a specific data layout for the values exchanged, so that they interpret them in the same way. This is in particular true for exchanging events with Zeek—which is one of the main applications for Broker in the first place. To support that, Broker provides built-in support for sending and receiving Zeek events through a small Zeek-specific shim on top of the generic message model. The shim encapsulates Zeek events and takes care of converting them into the expected lower-level message layout that gets transmitted. This way, Zeek events can be exchanged between an external Broker client and Zeek itself—and also even just between Broker clients without any Zeek instances at all.

Here's a complete ping/ping example between a C++ Broker client and Zeek:

```
# ping.zeek

redef exit_only_after_terminate = T;

global pong: event(n: int);

event ping(n: int)
        {
        event pong(n);
        }

event zeek_init()
        {
        Broker::subscribe("/topic/test");
        Broker::listen("127.0.0.1", 9999/tcp);
        Broker::auto_publish("/topic/test", pong);
        }
```

```
// ping.cc

#include <assert.h>

#include "broker/broker.hh"
#include "broker/zeek.hh"

using namespace broker;

int main() {
    // Setup endpoint and connect to Zeek.
    endpoint ep;
    auto sub = ep.make_subscriber({"/topic/test"});
    auto ss = ep.make_status_subscriber(true);
    ep.peer("127.0.0.1", 9999);

    // Wait until connection is established.
    auto ss_res = ss.get();
    auto st = caf::get_if<status>(&ss_res);
    if ( ! (st && st->code() == sc::peer_added) ) {
        std::cerr << "could not connect" << std::endl;
        return 1;
    }

    for ( int n = 0; n < 5; n++ ) {
        // Send event "ping(n)".
        zeek::Event ping("ping", {n});
        ep.publish("/topic/test", ping);

        // Wait for "pong" reply event.
        auto msg = sub.get();
        zeek::Event pong(move_data(msg));
        std::cout << "received " << pong.name() << pong.args() << std::endl;
    }

    return 0;
}
```

```
# g++ -std=c++11 -lbroker -lcaf_core -lcaf_io -lcaf_openssl -o ping ping.cc
# zeek ping.zeek &
# ./ping
received pong[0]
received pong[1]
received pong[2]
received pong[3]
received pong[4]
```

2.3 Data Model

Broker offers a data model that is rich in types, closely modeled after Zeek. Both *endpoints* and *data stores* operate with the `data` abstraction as basic building block, which is a type-erased variant structure that can hold many different values.

There exists a total ordering on `data`, induced first by the type discriminator and then its value domain. For a example, an `integer` will always be smaller than a `count`. While a meaningful ordering exists only when comparing two values of the same type, the total ordering makes it possible to use `data` as index in associative containers.

2.3.1 Types

None

The `none` type has exactly one value: `nil`. A default-construct instance of `data` is of type `none`. One can use this value to represent optional or invalid data.

Arithmetic

The following types have arithmetic behavior.

Boolean

The type `boolean` can take on exactly two values: `true` and `false`. A `boolean` is a type alias for `bool`.

Count

A `count` is a 64-bit *unsigned* integer and type alias for `uint64_t`.

Integer

An `integer` is a 64-bit *signed* integer and type alias for `int64_t`.

Real

A `real` is a IEEE 754 double-precision floating point value, also commonly known as `double`.

Time

Broker offers two data types for expressing time: `timespan` and `timestamp`.

Both types seamlessly interoperate with the C++ standard library time facilities. In fact, they are concrete specializations of the time types in `std::chrono`:

```
using clock = std::chrono::system_clock;
using timespan = std::chrono::duration<int64_t, std::nano>;
using timestamp = std::chrono::time_point<clock, timespan>;
```

Timespan

A `timespan` represents relative time duration in nanoseconds. Given that the internal representation is a 64-bit signed integer, this allows for representing approximately 292 years.

Timestamp

A `timestamp` represents an absolute point in time. The frame of reference for a `timestamp` is the UNIX epoch, January 1, 1970. That is, a `timestamp` is simply an anchored `timespan`. The function `now()` returns the current wallclock time as a `timestamp`.

String

Broker directly supports `std::string` as one possible type of `data`.

Enum Value

An `enum_value` wraps enum types defined by Zeek by storing the enum value's name as a `std::string`. The receiver is responsible for knowing how to map the name to the actual numeric value if it needs that information.

Networking

Broker comes with a few custom types from the networking domain.

Address

The type `address` is an IP address, which holds either an IPv4 or IPv6 address. One can construct an address from a byte sequence, along with specifying the byte order and address family. An `address` can be masked by zeroing a given number of bottom bits.

Subnet

A `subnet` represents an IP prefix in CIDR notation. It consists of two components: a network address and a prefix length.

Port

A `port` represents a transport-level port number. Besides TCP and UDP ports, there is a concept of an ICMP "port" where the source port is the ICMP message type and the destination port the ICMP message code.

Containers

Broker features the following container types: `vector`, `set`, and `table`.

Vector

A `vector` is a sequence of `data`.

It is a type alias for `std::vector<data>`.

Set

A `set` is a mathematical set with elements of type `data`. A fixed `data` value can occur at most once in a `set`.

It is a type alias for `std::set<data>`.

Table

A `set` is an associative array with keys and values of type `data`. That is, it maps `data` to `data`.

It is a type alias for `std::map<data, data>`.

2.3.2 Interface

The `data` abstraction offers two ways of interacting with the contained type instance:

1. Querying a specific type `T`. Similar to C++17's `std::variant`, the function `get_if<T>` returns either a `T*` if the contained type is `T` and `nullptr` otherwise:

   ```
   auto x = data{...};
   if (auto i = get_if<integer>(x))
     f(*i); // safe use of x
   ```

   Alternatively, the function `get<T>` returns a reference of type `T&` or `const T&`, based on whether the given `data` argument is const-qualified:

   ```
   auto x = data{...};
   auto& str = get<std::string>(x); // throws std::bad_cast on type clash
   f(str); // safe use of x
   ```

2. Applying a *visitor*. Since `data` is a variant type, one can apply a visitor to it, i.e., dispatch a function call based on the type discriminator to the active type. A visitor is a polymorphic function object with overloaded `operator()` and a `result_type` type alias:

```
struct visitor {
  using result_type = void;

  template <class T>
  result_type operator()(const T&) const {
    std::cout << ":-(" << std::endl;
  }

  result_type operator()(real r) const {
    std::cout << i << std::endl;
  }

  result_type operator()(integer i) const {
    std::cout << i << std::endl;
  }
};

auto x = data{42};
visit(visitor{}, x); // prints 42
x = 4.2;
visit(visitor{}, x); // prints 4.2
x = "42";
visit(visitor{}, x); // prints :-(
```

2.4 Data Stores

In addition to transmitting *data* via publish/subscribe communication, Broker also offers a mechanism to store this very data. Data stores provide a distributed key-value interface that leverages the existing *peer communication channels*.

2.4.1 Aspects

A data store has two aspects: a *frontend* for interacting with the user, and a *backend* that defines the database type for the key-value store.

Frontend

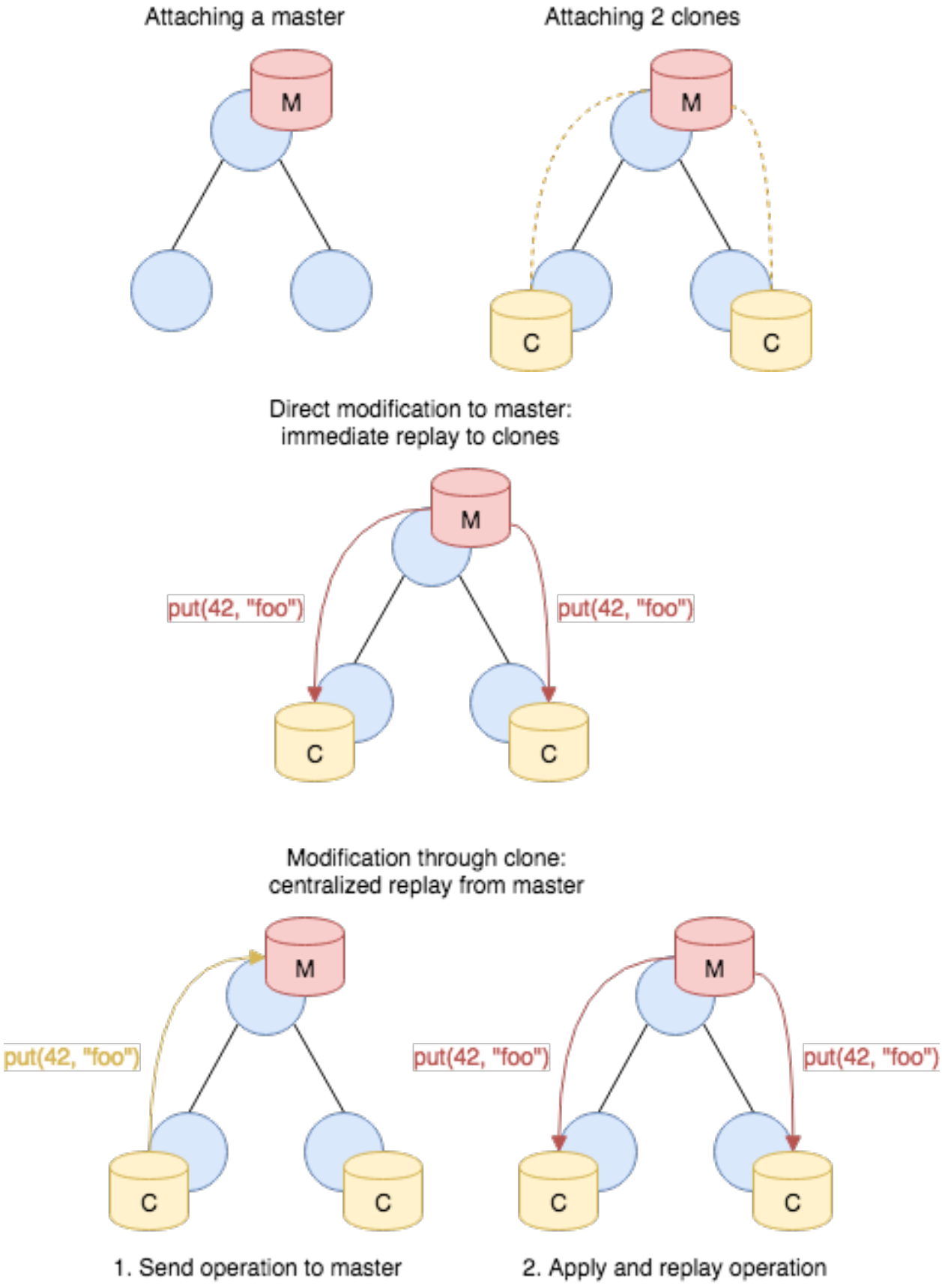
Users interact with a data store through the frontend, which is either a *master* or a *clone*. A master is authoritative for the store, whereas a clone represents a local cache that is connected to the master. A clone cannot exist without a master. Only the master can perform mutating operations on the store, which it pushes out to all its clones. A clone has a full copy of the data for faster access, but sends any modifying operations to its master first. Only when the master propagates back the change, the result of the operation becomes visible at the clone.

It is possible to attach one or more data stores to an endpoint, but each store must have a unique master name. For example, two peers cannot both have a master with the same name. When a clone connects to its master, it receives a full dump of the store:

While the master can apply mutating operations to the store directly, clones have to first send the operation to the master and wait for the replay for the operation to take on effect:

Backend

The master can choose to keep its data in various backends:


Attaching a master
Attaching 2 clones
M
M
C
C
Direct modification to master:
immediate replay to clones
M
put(42, "foo")
put(42, "foo")
C
C
Modification through clone:
centralized replay from master
M
M
put(42, "foo")
put(42, "foo")
put(42, "foo")
C
C
C
C
1. Send operation to master
2. Apply and replay operation

1. **Memory**. This backend uses a hash-table to keep its data in memory. It is the fastest of all backends, but offers limited scalability and does not support persistence.
2. SQLite. The SQLite backend stores its data in a SQLite3 format on disk. While offering persistence, it does not scale well to large volumes.
3. RocksDB. This backend relies on an industrial-strength, high-performance database with a variety of tuning knobs. If your application requires persistence and also needs to scale, this backend is your best choice.

2.4.2 Operations

Key operations on data stores include attaching it to an endpoint, performing mutating operations, and retrieving values at specific keys.

Construction

The example below illustrates how to attach a master frontend with a memory backend:

```
endpoint ep;
auto ds = ep.attach_master("foo", memory);
```

The factory function `endpoint::attach_master` has the following signature:

```
expected<store> attach_master(std::string name, backend type,
                              backend_options opts=backend_options());
```

The function takes as first argument the global name of the store, as second argument the type of store (`broker::{memory,sqlite,rocksdb}`), and as third argument optionally a set of backend options, such as the path where to keep the backend on the filesystem. The function returns a `expected<store>` which encapsulates a type-erased reference to the data store.

Note: The type `expected<T>` encapsulates an instance of type `T` or a `status`, with an interface that has "pointer semantics" for syntactic convenience:

```
auto f(...) -> expected<T>;

auto x = f();
if (x)
  f(*x); // use instance of type T
else
  std::cout << to_string(x.error()) << std::endl;
```

In the failure case, the `expected<T>::error()` holds an `error`.

Modification

Data stores support the following mutating operations:

`void put(data key, data value, optional<timespan> expiry = {}) const;` Stores the `value` at `key`, overwriting a potentially previously existing value at that location. If `expiry` is given, the new entry will automatically be removed after that amount of time.

`void erase(data key) const;` Removes the value for the given key, if it exists.

void clear() const; Removes *all* current store values.

void increment(data key, data amount, optional<timespan> expiry = {}) const; Increments the existing value at `key` by the given amount. This is supported for numerical data types and for timestamps. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

void decrement(data key, data amount, optional<timespan> expiry = {}) const; Decrements the existing value at `key` by the given amount. This is supported for numerical data types and for timestamps. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

void append(data key, data str, optional<timespan> expiry = {}) const; Appends a new string `str` to an existing string value at `key`. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

void insert_into(data key, data index, optional<timespan> expiry = {}) const; For an existing set value stored at `key`, inserts the value `index` into it. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

void insert_into(data key, data index, data value, optional<timespan> expiry = {}) const; For an existing vector or table value stored at `key`, inserts `value` into it at `index`. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

void remove_from(data key, data index, optional<timespan> expiry = {}) const; For an existing vector, set or table value stored at `key`, removes the value at `index` from it. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

void push(data key, data value, optional<timespan> expiry = {}) const; For an existing vector at `key`, appends `value` to its end. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

void pop(data key, optional<timespan> expiry = {}) const; For an existing vector at `key`, removes its last value. If `expiry` is given, the modified entry's expiration time will be updated accordingly.

Direct Retrieval

Data stores support the following retrieval methods:

expected<data> get(data key) const; Retrieves the value at `key`. If the key does not exist, returns an error `ec::no_such_key`.

```
auto result = ds->get("foo");
if (result)
    std::cout << *result << std::endl; // Print current value of 'foo'.
else if (result.error() == ec::no_such_key)
    std::cout << "key 'foo' does not exist'" << std::endl;
else if (result.error() == ec::backend_failure)
    std::cout << "something went wrong with the backend" << std::endl;
else
    std::cout << "could not retrieve value at key 'foo'" << std::endl;
```

expected<data> exists(data key) const; Returns a `boolean` data value indicating whether `key` exists in the store.

expected<data> get_index_from_value(data key, data index) const; For containers values (sets, tables, vectors) at `key`, retrieves a specific `index` from the value. For sets, the returned value is a `boolean` data instance indicating whether the index exists in the set. If `key` does not exist, returns an error `ec::no_such_key`.

expected<data> keys() const Retrieves a copy of all the store's current keys, returned as a set. Note that this is a potentially expensive operation if the store is large.

All of these methods may return the `ec::stale_data` error when querying a clone if it has yet to ever synchronize with its master or if has been disconnected from its master for too long of a time period. The length of time before a clone's cache is deemed stale depends on an argument given to the `endpoint::attach_clone` method.

All these methods share the property that they will return the corresponding result directly. Due to Broker's asynchronous operation internally, this means that they may block for short amounts of time until the result becomes available. If that's a problem, you can receive results back asynchronously as well, see next section.

Note, however, that even with this direct interface, results may sometimes take a bit to reflect operations that clients perform (including the same client!). This effect is most pronounced when working through a clone: any local manipulations will need to go through the master before they become visible to the clone.

Proxy Retrieval

When integrating data store queries into an event loop, the direct retrieval API may not prove a good fit: request and response are coupled at lookup time, leading to potentially blocking operations. Therefore, Broker offers a second mechanism to lookup values in data stores. A `store::proxy` decouples lookup requests from responses and exposes a *mailbox* to integrate into event loops. When a using a proxy, each request receives a unique, monotonically increasing 64-bit ID that is hauled through the response:

```
// Add a value to a data store (master or clone).
ds->put("foo", 42);
// Create a proxy.
auto proxy = store::proxy{*ds};
// Perform an asynchyronous request to look up a value.
auto id = proxy.get("foo");
// Get a file descriptor for event loops.
auto fd = proxy.mailbox().descriptor();
// Wait for result.
::pollfd p = {fd, POLLIN, 0};
auto n = ::poll(&p, 1, -1);
if (n < 0)
    std::terminate(); // poll failed

if (n == 1 && p.revents & POLLIN) {
    auto response = proxy.receive(); // Retrieve result, won't block now.
    assert(response.id == id);
    // Check whether we got data or an error.
    if (response.answer)
        std::cout << *response.answer << std::endl; // may print 42
    else if (response.answer.error() == ec::no_such_key)
        std::cout << "no such key: 'foo'" << std::endl;
    else
        std::cout << "failed to retrieve value at key 'foo'" << std::endl;
}
```

The proxy provides the same set of retrieval methods as the direct interface, with all of them returning the corresponding ID to retrieve the result once it has come in.

2.5 Python Bindings

Almost all functionality of Broker is also accessible through Python bindings. The Python API mostly mimics the C++ interface, but adds transparent conversion between Python values and Broker values. In the following we demonstrate the main parts of the Python API, assuming a general understanding of Broker's concepts and the C++ interface.

Note: Broker's Python bindings require Python 2.7 or Python 3. If you are using Python 2.7, then you will need to install the ipaddress module from PyPI (one way to do this is to run "pip install ipaddress").

2.5.1 Installation in a Virtual Environment

To install Broker's python bindings in a virtual environment, the **python-prefix** configuration option can be specified and the python header files must be on the system for the version of python in the virtual environment. You can also use the **prefix** configuration option to install the main Broker library and headers into an isolated location.

```
$ virtualenv -p python3 /Users/user/sandbox/broker/venv
$ . /Users/user/sandbox/broker/venv/bin/activate
$ ./configure --prefix=/Users/user/sandbox/broker --python-prefix=$(python -c 'import␣
↪sys; print(sys.exec_prefix)')
$ make install
$ python -c 'import broker; print(broker.__file__)'
/Users/user/sandbox/broker/venv/lib/python3.7/site-packages/broker/__init__.py
```

2.5.2 Communication

Just as in C++, you first set up peerings between endpoints and create subscriber for the topics of interest:

```
ep1 = broker.Endpoint()
ep2 = broker.Endpoint()

s1 = ep1.make_subscriber("/test")
s2 = ep2.make_subscriber("/test")

port = ep1.listen("127.0.0.1", 0)
ep2.peer("127.0.0.1", port, 1.0)
```

You can then start publishing messages. In Python a message is just a list of values, along with the corresponding topic. The following publishes a simple message consisting of just one string, and then has the receiving endpoint wait for it to arrive:

```
ep2.publish("/test", ["ping"])
(t, d) = s1.get()
# t == "/test", d == ["ping"]
```

Example of publishing a small batch of two slightly more complex messages with two separate topics:

```
msg1 = ("/test/2", (1, 2, 3))
msg2 = ("/test/3", (42, "foo", {"a": "A", "b": ipaddress.IPv4Address('1.2.3.4
↪')}))
ep2.publish_batch(msg1, msg2)
```

As you see with the 2nd message there, elements can be either standard Python values or instances of Broker wrapper classes; see the data model section below for more.

The subscriber instances have more methods matching their C++ equivalent, including `available` for checking for pending messages, `poll()` for getting available messages without blocking, `fd()` for retrieving a select-able file descriptor, and `{add,remove}_topic` for changing the subscription list.

2.5.3 Exchanging Zeek Events

The Broker Python bindings come with support for representing Zeek events as well. Here's the Python version of the *C++ ping example shown earlier*:

```
# ping.zeek

redef exit_only_after_terminate = T;

global pong: event(n: int);

event ping(n: int)
        {
        event pong(n);
        }

event zeek_init()
        {
        Broker::subscribe("/topic/test");
        Broker::listen("127.0.0.1", 9999/tcp);
        Broker::auto_publish("/topic/test", pong);
        }
```

```
# ping.py

import sys
import broker

# Setup endpoint and connect to Zeek.
ep = broker.Endpoint()
sub = ep.make_subscriber("/topic/test")
ss = ep.make_status_subscriber(True);
ep.peer("127.0.0.1", 9999)

# Wait until connection is established.
st = ss.get()

if not (type(st) == broker.Status and st.code() == broker.SC.PeerAdded):
    print("could not connect")
    sys.exit(0)

for n in range(5):
    # Send event "ping(n)".
    ping = broker.zeek.Event("ping", n);
    ep.publish("/topic/test", ping);

    # Wait for "pong" reply event.
    (t, d) = sub.get()
    pong = broker.zeek.Event(d)
    print("received {}{}".format(pong.name(), pong.args()))
```

```
# python3 ping.py
received pong[0]
received pong[1]
received pong[2]
received pong[3]
received pong[4]
```

2.5.4 Data Model

The Python API can represent the same type model as the C++ code. For all Broker types that have a direct mapping to a Python type, conversion is handled transparently as values are passed into, or retrieved from, Broker. For example, the message `[1, 2, 3]` above is automatically converted into a Broker list of three Broker integer values. In cases where there is not a direct Python equivalent for a Broker type (e.g., for `count`; Python does not have an unsigned integer class), the Broker module provides wrapper classes. The following table summarizes how Broker and Python values are mapped to each other:

Broker Type	Python representation
`boolean`	`True/False`
`count`	`broker.Count(x)`
`integer`	`int`
`real`	`float`
`timespan`	`datetime.timedelta`
`timestamp`	`datetime.datetime`
`string`	`str`
`address`	`ipaddress.IPv4Address/ipaddress.IPv6Address`
`subnet`	`ipaddress.IPv4Network/ipaddress.IPv6Network`
`port`	`broker.Port(x, broker.Port.{TCP,UDP,ICMP,Unknown})`
`vector`	`tuple`
`set`	`set`
`table`	`dict`

Note that either a Python `tuple` or Python `list` may convert to a Broker `vector`, but the canonical Python type representing a `vector` is a tuple. That is, whenever converting a Broker `vector` value into a Python value, you will get a `tuple`. A `tuple` is the canonical type here because it is an immutable type, but a `list` is mutable – we need to be able to represent tables indexed by vectors, tables are mapped to Python dictionaries, Python dictionaries only allow immutable index types, and so we must use a `tuple` to represent a `vector`.

2.5.5 Status and Error Messages

Status and error handling works through a status subscriber, again similar to the C++ interface:

```
ep1 = broker.Endpoint()
es1 = ep1.make_status_subscriber()
r = ep1.peer("127.0.0.1", 1947, 0.0) # Try unavailable port, no retry
st1 = es1.get()
# s1.code() == broker.EC.PeerUnavailable
```

```
ep1 = broker.Endpoint()
ep2 = broker.Endpoint()
es1 = ep1.make_status_subscriber(True)
es2 = ep2.make_status_subscriber(True)
port = ep1.listen("127.0.0.1", 0)
ep2.peer("127.0.0.1", port, 1.0)
st1 = es1.get()
st2 = es2.get()
# st1.code() == broker.SC.PeerAdded, st2.code() == broker.SC.PeerAdded
```

2.5.6 Data Stores

For data stores, the C++ API also directly maps to Python. The following instantiates a master store to then operate on:

```
ep1 = broker.Endpoint()
m = ep1.attach_master("test", broker.Backend.Memory)
m.put("key", "value")
x = m.get("key")
# x == "value"
```

In Python, both master and clone stores provide all the same accessor and mutator methods as C++. Some examples:

```
m.increment("e", 1)
m.decrement("f", 1)
m.append("str", "ar")
m.insert_into("set", 3)
m.remove_from("set", 1)
m.insert_into("table", 3, "D")
m.remove_from("table", 1)
m.push("vec", 3)
m.push("vec", 4)
m.pop("vec")
```