Broker

version 0.5.0

The Bro Project

January 16, 2017

Contents

Broker User Manual

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

Outline

Section 1 introduces Broker's key components and basic terminology, such as *contexts*, *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 und publish under a given topic. The same data also works with 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*: in-memory, SQLite, and RocksDB.

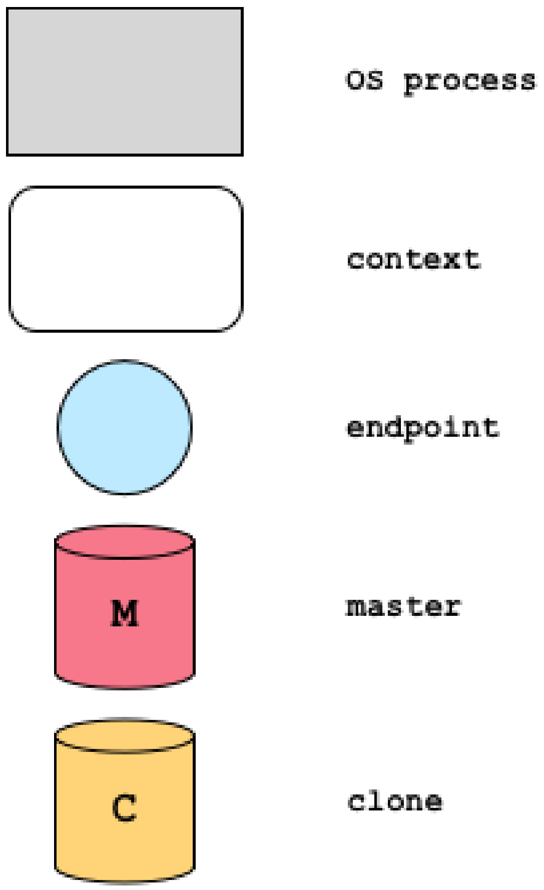
Synopsis

```
using namespace broker;

context ctx;
auto ep = ctx.spawn<nonblocking>(); // create an endpoint
ep.peer(1.2.3.4, 42000); // peer with a remote endpoint
ep.publish("/foo", set{1, 2, 3}); // publish data under a given topic
ep.subscribe("/foo", [=](const topic& t, const data& d) {
  std::cout << "got data for topic " << t << ": " << d << std::endl;
});

auto m = ep.attach<master, rocksdb>("yoda", "/tmp/database");
m.put(4.2, -42);
m.put("bar", vector{true, 7u, time::now()});
m.get<nonblocking>(4.2).then(
  [=](const data& d) {
    process(d);
  },
  [=](status s) {
    if (s == ec::key_not_found)
      std::cout << "no such key: 4.2" << std::endl;
    else
      std::terminate();
  }
);
```

Overview

The **Broker** library enables applications to communicate in Bro'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.

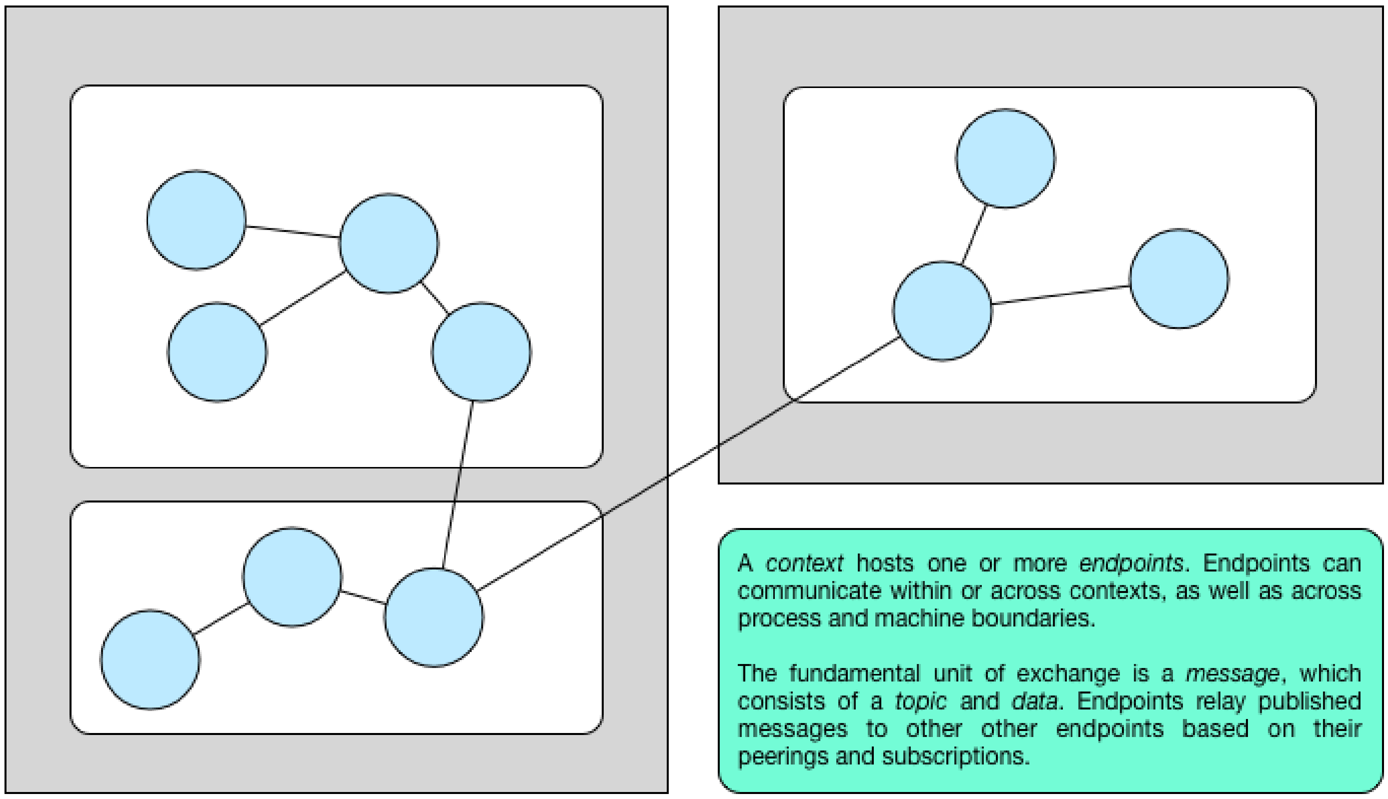


Moreover, all C++ code examples assume `using namespace broker` for conciseness.

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 *context* hosts one or more *endpoints*. Endpoints can communicate within or across contexts, as well as across process and machine boundaries.

The fundamental unit of exchange is a *message*, which consists of a *topic* and *data*. Endpoints relay published messages to other other endpoints based on their peerings and subscriptions.

There exist two API flavors: a synchronous (blocking) and asynchronous (non-blocking) version. Internally, Broker operates entirely asynchronously by leveraging the C++ Actor Framework (CAF). Users can receive messages either by calling a blocking function call until the next message arrives, or by installing a callback for a specific topic.

See Section 2 for concrete usage examples.

Data Model

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

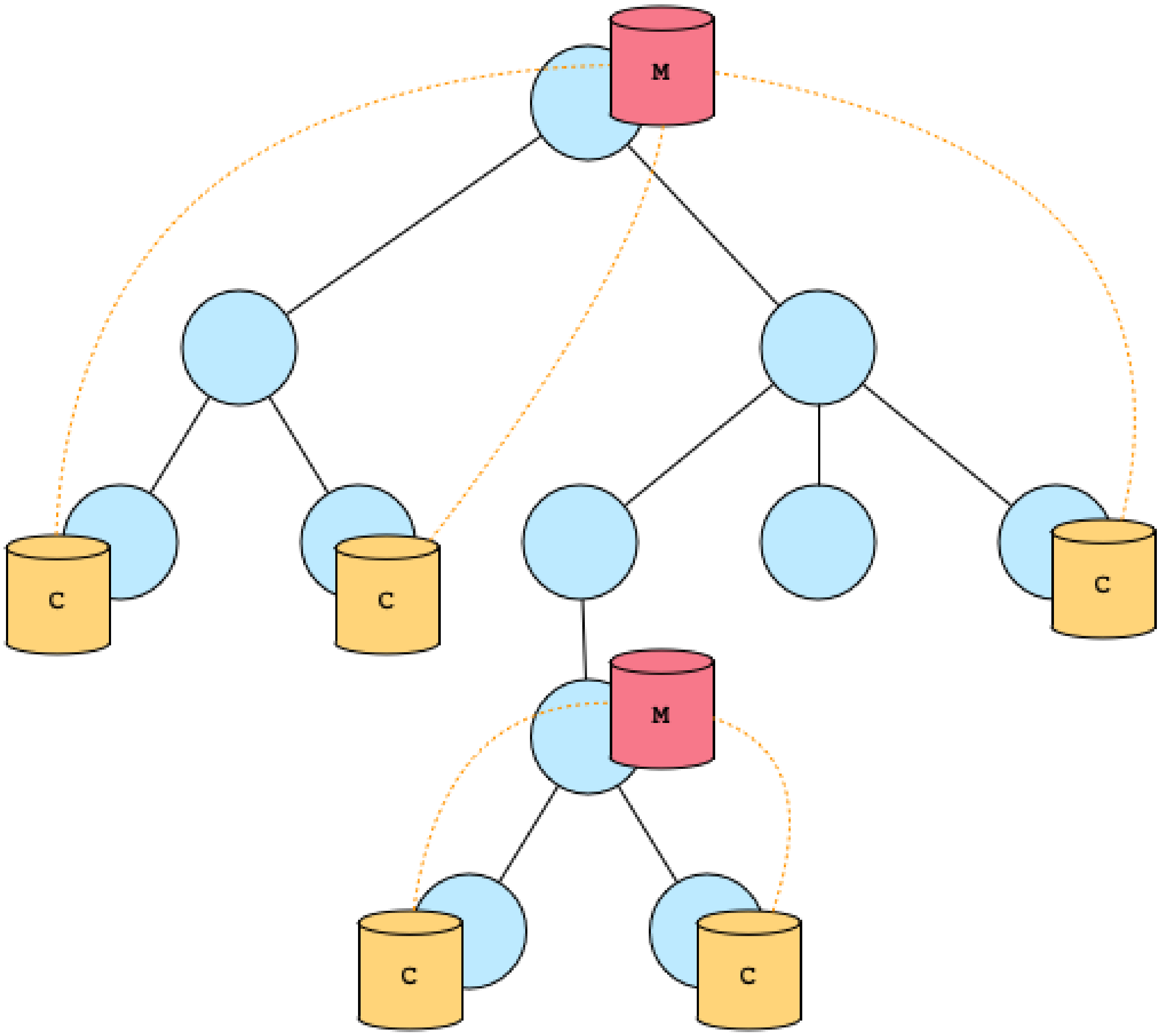
- `none`
- `boolean`
- `count`
- `integer`
- `real`
- `timespan`
- `timestamp`
- `string`
- `address`
- `subnet`
- `port`
- `vector`
- `set`

- `table`

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

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 represent 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 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: in-memory, SQLite, and RocksDB.

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

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 a topic of interest. The asynchronous nature of publish/subscribe makes it a popular choice for loosely coupled, distributed systems.

Broker is the successor of Broccoli, Bro's client communications library. Broccoli enables arbitrary applications to communicate in Bro's data model.

Endpoints

Broker encapsulates its entire state in a `context` object. Multiple instances of a `context` can exist in the same process, but each `context` features a thread-pool and (configurable) scheduler, which determines the execution of Broker's components. Using a single `context` 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.

A `context` can *spawn* `endpoint` instances, of which there exists a *blocking* and *non-blocking* variant. The two types differ in the way they manage their subscriptions and receive messages. Both variants have a *mailbox*, which is essentially a queue with its unprocessed messages. In the blocking case, the user manually extracts messages from the mailbox, whereas the Broker runtime dispatches messages asynchronously in the non-blocking case.

Both endpoint variants can be mixed and matched, it is not necessary to commit to a particular type for all endpoints within a context.

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 thread-safe.

Receiving Data

Endpoints can receive data through an explicit call to `receive` (blocking API) or installing a callback invoked by the runtime (non-blocking API).

Blocking

The blocking API exists for applications that primarily operate synchronously and/or ship their own event loop. Endpoints subscribe to various topics and call a `receive` function to block and wait for message:

```
context ctx;
auto ep = ctx.spawn<blocking>();
ep.subscribe("foo");
auto msg = ep.receive(); // block and wait until a message arrives
if (msg)
  std::cout << msg.topic() << " -> " << msg.data() << std::endl;
else
  std::cout << to_string(msg.status()) << std::endl;
```

The function `receive` blocks until the endpoint receives a `message`, which is either a (`topic`, `data`) pair or a `status` to signal an error or a status change of the endpoint topology. More on `status` messages in Section 2.3.

Blocking indefinitely does not work well in combination with existing event loops or polling. Therefore, blocking endpoints offer an additional `mailbox` API:

```
// Manual polling.
if (!ep.mailbox().empty()) {
  auto msg = ep.receive(); // guaranteed to not block
  ...
}

// Introspection.
auto n = ep.mailbox().count(); // O(n) due to internal queue implementation

// Event loop integration.
auto fd = ep.mailbox().descriptor();
::pollfd p = {fd, POLLIN, {}};
auto n = ::poll(&p, 1, timeout);
if (n < 0)
  std::terminate(); // poll failed
if (n == 1 && p.revents & POLLIN) {
  auto msg = ep.receive(); // guaranteed to not block
  ...
}
```

Non-Blocking

If your application does not require a blocking API, the non-blocking API offers an asynchronous alternative. Unlike the blocking API, non-blocking endpoints take a callback for each topic they subscribe to:

```
context ctx;
auto ep = ctx.spawn<nonblocking>();
ep.subscribe("/foo", [=](const topic& t, const data& d) {
  std::cout << t << " -> " << d << std::endl;
});
ep.subscribe("/bar", [=](const topic& t, const data& d) {
  std::cout << t << " -> " << d << std::endl;
});
```

When a new message matching the subscription arrives, Broker dispatches it to the callback without blocking.

Warning

The function `subscribe` returns immediately. Capturing variable *by reference* introduces a dangling reference once the outer frame returns. Therefore, only capture locals *by value*.

Sending Data

The API for sending data is the same for blocking and non-blocking endpoints. In Broker, a *message* is a *topic-data* pair. That is, endpoints *publish* data under a *topic* to send a message to all subscribers:

```
ep.publish("foo", 42);
ep.publish("bar", vector{1, 2, 3});
ep.publish("baz", 1, 2, 3); // same as above, implicit conversion to vector
```

The one-argument version of `publish` takes as first argument a `topic` and `data` instance. The variadic version implicitly constructs a `vector` from the provided `data` instances.

Note

Publishing a message can be 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.

See Section 3 for a detailed discussion on how to construct various types of `data`.

Peerings

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

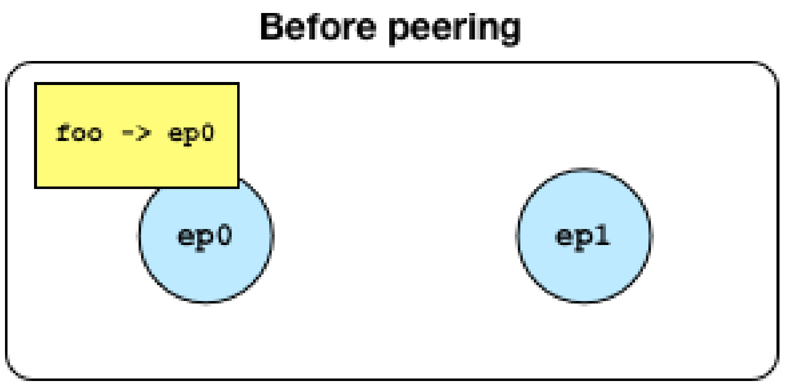
Note

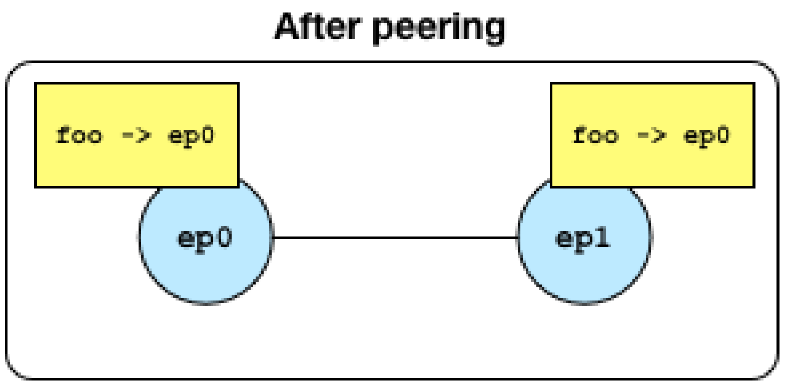
Broker currently does not support topologies with loops. This is purely a technical limitation and vanishes in the future.

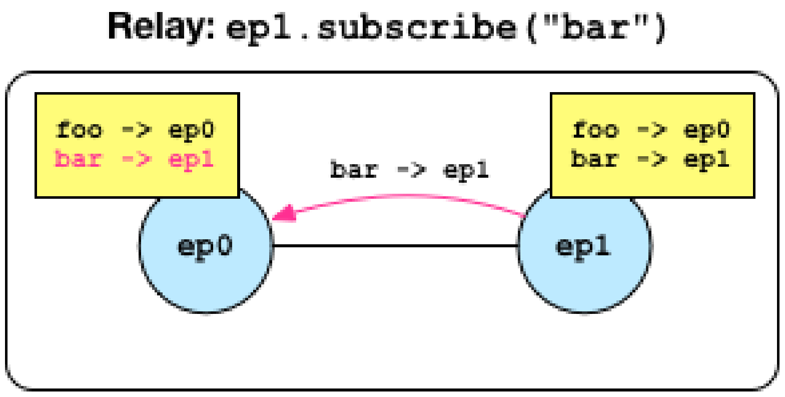
An endpoint can either peer with a local or a remote endpoint:

```
context ctx;
auto ep0 = ctx.spawn<blocking>();
ep0.subscribe("foo");
auto ep1 = ctx.spawn<nonblocking>();
ep0.peer(ep1); // exchanges existing subscriptions
ep1.subscribe("bar"); // relays subscription to peers
```

The figure below shows the subscription before and after entering the peering relationship.



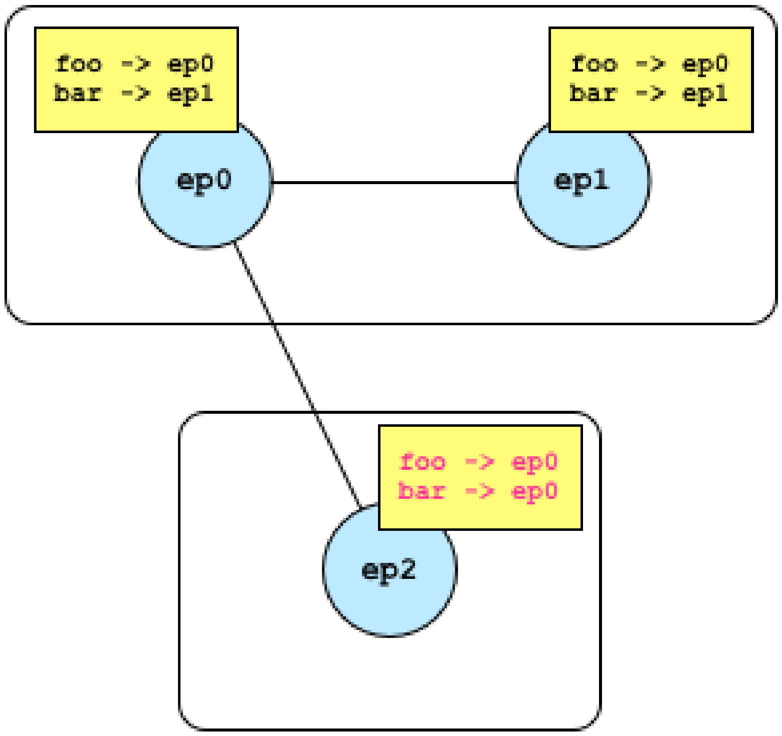




Let's consider a third endpoint joining, this time through a remote connection.

```
// Expose endpoint at an IP address and TCP port.
ep0.listen(1.2.3.4, 40000);

// In a separate OS process, connect to it.
context ctx;
auto ep2 = ctx.spawn<nonblocking>();
ep2.peer(1.2.3.4, 42000); // installs a remote peering
```

Thereafter, we have the following topology:



Note that `ep2` does not know about `ep1` and forwards `data` for topic `foo` and `bar` via `ep0`. However, `ep2.publish("bar", 42)` still forwards a message via `ep0` to `ep1`.

Status Messages

Broker presents errors and runtime changes to the user as `status` messages. Blocking endpoints obtain them via `receive` and non-blocking endpoints must subscribe to them explicitly. For example, after a successful peering, both endpoints receive a `peer_added` status message:

```
context ctx;
// Peer two endpoints.
auto ep0 = ctx.spawn<blocking>();
auto ep1 = ctx.spawn<blocking>();
ep0.peer(ep1);
// Block and wait for status messages.
auto msg0 = ep0.receive();
assert(!msg0);
if (msg0.status() == sc::peer_added)
  std::cout << "peering established successfully" << std::endl;
```

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 = 1,
  /// Successfully added a new peer.
  peer_added,
  /// Successfully removed a peer.
  peer_removed,
  /// Version incompatibility.
  peer_incompatible,
  /// Referenced peer does not exist.
  peer_invalid,
  /// Remote peer not listening.
  peer_unavailable,
  /// An peering request timed out.
  peer_timeout,
  /// Lost connection to peer.
  peer_lost,
  /// Re-gained connection to peer.
  peer_recovered,
  /// 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,
};
```

Status messages have an optional *context* and an optional descriptive *message*:

```
auto& s = msg.status();
// Check for textual description.
if (auto str = s.message())
  std::cout << *str << std::endl;
// Check for contextual information.
if (auto ctx = s.context<endpoint_info>())
  if (ctx->network)
    std::cout << "peer at: "
              << ctx->network->address << ':'
              << ctx->network->port << std::endl;
```

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 `enpoint_info` context as well as a message.

Non-blocking endpoints ignore `status` messages unless they subscribe to them explicitly:

```
context ctx;
auto ep = ctx.spawn<nonblocking>();
ep.subscribe(
  [&](const status& s) { ... }
);
```

Data Model

Broker offers a data model that is rich in types, closely modeled after Bro. 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`. Only when comparing two values of the same type, a meaningful ordering exists. The total ordering makes it possible to use `data` as index in associative containers.

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

A `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 seemlessly interoperate with the C++ standard library time faciliates. In fact, they are concrete specializations of the time types in `std::chrono`:

```
namespace broker {

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

timestamp now();

} // namespace broker
```

Timespan

An `timespan` represents relative time duration in nanoseconds. Given that the internal reprsentation 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`.

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 mathemtical 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 associative array with keys and values of typ `data`. That is, it maps `data` to `data`.

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

Interface

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

1. Querying a specific type `T`. The function `data::get<T>` returns either a `T*` if the contained type is `T` and `nullptr` otherwise:

   ```
   auto x = data{...};
   if (auto i = x.get<integer>())
     f(*i); // 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 discrimantor 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 :-(
```

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 leverage the existing peer communication channels.

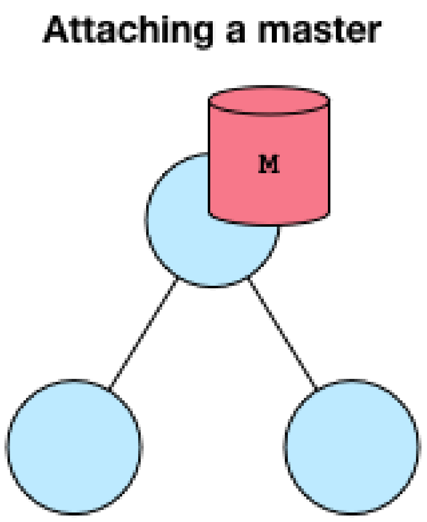
Aspects

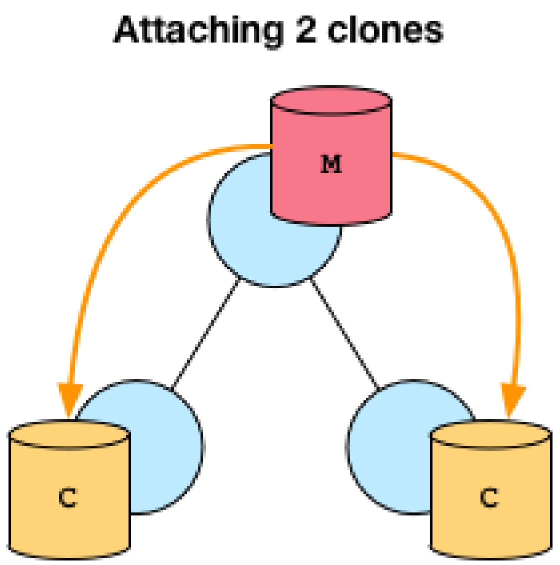
A data store has two aspects: a *frontend* for interacting with the user, and a *backend* that defines the database type for the key-value store.

Frontend

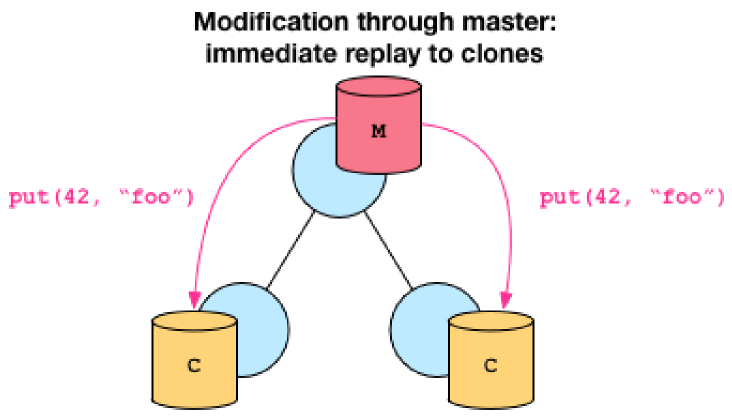
Users interact with a data store through the frontend, which is either a *master* or a *clone*. A master is authoritative for the store, whereas a clone represents a local cache that is connected to the master. A clone cannot exist without a master. Only the master can perform mutating operations on 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.

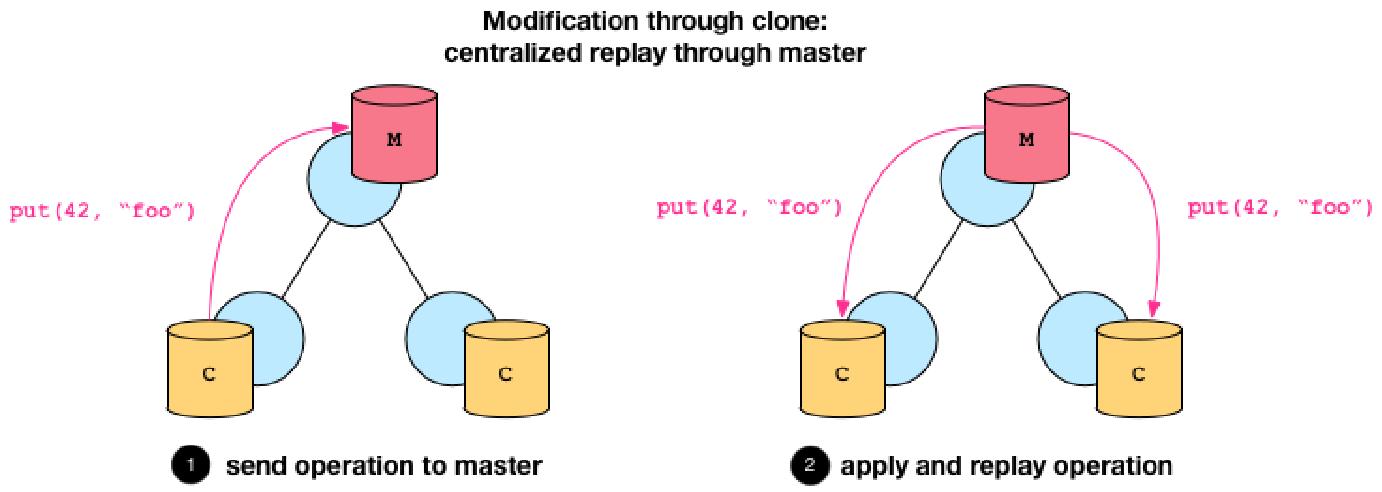
It is possible to attach one or more data stores to an endpoint, but each store must have a unique master name. For example, two peers cannot both have a master with the same name. When a clone connects to its master, it receives full dump of the store:





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:




Modification through clone:
centralized replay through master
M
put(42, "foo")
C
C
1 send operation to master
M
put(42, "foo")
put(42, "foo")
C
C
2 apply and replay operation

Backend

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

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 (currently) 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 very well.
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.

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:

```
context ctx;
auto ep = ctx.spawn<nonblocking>();
// attach a master with memory backend
auto ds = ep.attach<master, memory>("foo");
if (ds)
```

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

```
attach<F, B>(const std::string& name, backend_options opts = {})
```

The two template parameters `F` and `B` denote the respective frontend and backend types, where `B` defaults to `memory`. The function takes as first argument the name of the store and as second argument optionally a set of backend options, such as the path where to keep the backend on the filesystem. The function returns a `result<store>` which encapsulates a type-erased reference to the data store.

Note

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

```
auto f(...) -> result<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 `result<T>::status()` holds a `status` that can be compared against the status code enumeration `sc`.

Modification

Data stores support the following mutating operations:

1. `put(key, value)`: stores `value` at `key`, overwriting a potentially previously existing value at `key`.

```
ds.put(42, set{1, 2, 3});
ds.put("foo", 4.2);
```

2. `erase(key)`: removes the value for the given key, if `key` exists

```
ds.erase(42);    // removes set{1, 2, 3}, which got inserted above
ds.erase("bar"); // nop: key does not exist
```

> **Note**
>
> TODO: Document type-specific `add` and `remove`.

Direct Retrieval

There exist two methods of directly extracting values from a store: either in a blocking or non-blocking fashion.

The overload `get<blocking>(const data& key)` retrieves a value in a blocking manner and returns an instance of `result<data>`.

```
auto result = ds->get<blocking>("foo");
if (result)
  std::cout << *result << std::endl; // may print 4.2
else if (result.error() == sc::no_such_key)
  std::cout << "key 'foo' does not exist'" << std::endl;
else if (result.error() == sc::backend_failure)
  std::cout << "something went wrong with the backend" << std::endl;
else
  std::cout << "could not retrieve value at key 'foo'" << std::endl;
```

The overload `get<nonblocking>(const data& key)` returns a future-like proxy object which has the sole purpose of invoking `.then(...)` on it to install a one-shot handler that the runtime executes as soon as the result of the retrieval operation is available.

```
ds->get<nonblocking>("foo").then(
  [=](const data& d) {
    std::cout << d << std::endl; // may print 4.2
  },
  [=](const error& e) {
    if (result.error() == sc::no_such_key)
      std::cout << "key 'foo' does not exist'" << std::endl;
    else if (result.error() == sc::backend_failure)
      std::cout << "something went wrong with the backend" << std::endl;
    else
      std::cout << "could not retrieve value at key 'foo'" << std::endl;
  }
);
```

Proxy Retrieval

When integrating data store queries into an event loop, the direct retrieval API does not prove a good fit: there's no descriptor that we can poll, and request and response are coupled at lookup time. 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---exactly like blocking endpoints. Each request has 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().fd();
// Receive results or block until the result is available.
auto response = proxy.receive();
assert(response.id == id)
// Check whether we got data or an error.
if (response.answer)
  std::cout << *result.answer << std::endl; // may print 42
else if (response.answer.status() == sc::no_such_key)
  std::cout << "no such key: 'foo'" << std::endl;
else
  std::cout << "failed to retrieve value at key 'foo'" << std::endl;
```