Communication in Bro

- **2005**: Broccoli, Independent State
- **2007**: Bro Cluster
- **2008**: Python Bindings Broccoli
- **2011**: Ruby & Perl Bindings Broccoli
in the context of a single process is a minor subset of the NIDS process’s full state: either higher-level results (often just alerts) sent between processes to facilitate correlation or aggregation, or log files written to disk for processing in the future. The much richer (and bulkier) internal state of the NIDS remains exactly that, internal. It cannot be accessed by other processes unless a special means is provided for doing so, and it is permanently lost upon termination of the process. As a result, much of the management of the state is not only difficult but also not possible.

In this work we highlight the power of independent state, i.e., internal fine-grained state that can be propagated from one instance of a NIDS to others running either concurrently or sequentially. Independent state provides us with a wealth of capabilities of NIDSs. We discuss an implementation of independent state using the full set of mechanisms provided by the Bro NIDS (which, due to a crash, may happen unexpectedly). Independent state can be propagated from one instance of a NIDS to other, concurrently executing, instances. Considering two basic types of independent state. The first is particularly important: by keeping fine-grained state, rather than only aggregated state such as alerts or activity summaries, we can continue to process the independent state, even in the face of crashes. This is especially useful in intrusion detection systems, where information about previous activity is critical to analyzing current activity. The goal is to enable much of the semantically rich, detailed state to continue to exist after an instance of a NIDS has exited. For both types of state, the state essentially exists “outside” of an event engine or policy script executing process to become independent of that process. We call the state that continues to exist after an instance of a NIDS has exited “further independent” state. This state can be shared between processes, or accessed over time—that already appears in numerous existing systems—via bindings.

Independent state is a key to many of the possible applications that hold promise for enhancing the usefulness of Network Intrusion Detection Systems (NIDSs) critically rely on processing a great deal of state. Often much of this state resides solely in the volatile processor memory accessible to a single user-level process on a single machine. In this way—that already appears in numerous existing systems—can be shared between processes or accessed over time. Generally, any state that is maintained across the lifetime of a process, and that can be accessed by other processes unless a special means is provided for doing so, and it is permanently lost upon termination of the process. As a result, much of the management of the state is not only difficult but also not possible.
Outline

- Overview
- API
- Performance
- Outlook
Overview
Broker = Bro'ish data model

+ publish/subscribe communication

+ distributed key-value stores
Publish/Subscribe Communication
Publish/Subscribe Communication
Publish/Subscribe Communication

Internet

Organization

C++

Model
Publish/Subscribe Communication

Internet

Organization

Model
Model
Model
Publish/Subscribe Communication

Internet

Organization
Publish/Subscribe Communication
Publish/Subscribe Communication

Internet

Organization

File

C++

File

File
Publish/Subscribe Communication
Publish/Subscribe Communication
Publish/Subscribe Communication
Distributed Key-Value Stores
API
Lessons Learned

• **Functionality**: It Just Works

• **Usability**: no native type support, lots of "data wrapping"

• **Semantics**: no support for nonblocking processing
using namespace broker;

init();

endpoint ep{"sender"};
ep.peer("127.0.0.1", 9999);
ep.outgoing_connection_status().need_pop();

auto msg = message{
    "my_event",
    "Hello C++ Broker!",
    42u
};
ep.send("bro/event", msg);
ep.outgoing_connection_status().need_pop();

Current API

Initialize the Broker library.  
(Only one broker instance per process allowed.)

Create a local endpoint.

Block until connection status changes.

When communicating with Bro, the first argument must be a string identifying the event name. The remaining values represent the event arguments.

Publish the event under topic bro/event.

Block until connection status changes.
using namespace broker;

context ctx;

auto ep = ctx.spawn<blocking>();
ep.peer("127.0.0.1", 9999);

auto v = vector{
    "my_event",
    "Hello C++ Broker!",
    42u
};

ep.publish("bro/event", v);

A `context` encapsulates global state for a set of endpoints (e.g., worker threads, scheduler, etc.)

Create a local endpoint with `blocking` API.

Create a vector of data.
New semantics: a `message` is a `topic` plus `data`, not a sequence of data.

Publish the event under topic `bro/event`. 
Blocking vs. Non-Blocking API

context ctx;
auto ep = ctx.spawn<blocking>();

ep.subscribe("foo");
ep.subscribe("bar");

// Block and wait.
auto msg = ep.receive();
cout << msg.topic()
  << " - > "
  << msg.data()
  << endl;

// Equivalent semantics; functional API.
ep.receive(
  [&](const topic& t, const data& d) {
    cout << t << " - > " << d << endl;
  }
);

context ctx;
auto ep = ctx.spawn<nonblocking>();

// Called asynchronously by the runtime.
ep.subscribe(
  "foo",
  [=](const topic& t, const data& d) {
    cout << t << " - > " << d << endl;
  }
);

// As above, just for a different topic.
ep.subscribe(
  "bar",
  [=](const topic& t, const data& d) {
    cout << t << " - > " << d << endl;
  }
);
Data Store APIs

// Setup endpoint topology.
context ctx;
auto ep0 = ctx.spawn<blocking>();
auto ep1 = ctx.spawn<blocking>();
auto ep2 = ctx.spawn<blocking>();
ep0.peer(ep1);
ep0.peer(ep2);
// Attach stores.
auto m = ep0.attach<master<memory>("lord");
auto c0 = ep1.attach<clone>("lord");
auto c1 = ep2.attach<clone>("lord");
// Write to the master directly.
m->put("foo", 42);
m->put("bar", "baz");
// After propagation, query the clones.
sleep(propagation_delay);
auto v0 = c0->get("key");
auto v1 = c1->get("key");
assert(v0 && v1 && *v0 == *v1);

Available backends:
1. In-memory
2. SQLite
3. RocksDB
Data Store APIs

// Blocking API. Returns expected<data>.
auto v = c->get<blocking>("key");

// Non-blocking API.
// Runtime invokes callback.
c->get<nonblocking>("key").then(
  [=](data& d) {
    cout << "got it: " << d << endl;
  },
  [=](error& e) {
    cerr << "uh, this went wrong: "
        << e
        << endl;
  };
);
Performance
Simple Benchmark

• Throughput analysis

• Two endpoints: sender & receiver

• Message = conn.log entry

• System: MacBook Pro

• 16 GB RAM

• 4 x 2.8 GHz Core i7
Throughput

Throughput (msg/sec)

Version

new

old

Throughput

40K

20K

0

40%

new

old
Outlook
Roadmap to 1.0

1. Finish Python bindings
2. Implement Bro endpoint
3. Pattern matching in Bro
4. Flow control

```javascript
function lookup(key: string) : any;

when ( local x = lookup("key") )
{
  local result = "";
  switch( x )
  {
    case addr:
      if ( x in 10.0.0.0/8 )
        result = "contained";
    case string:
      result = "error: lookup() failed: " + x;
  }
}
```
Flow Control
Flow Control

Intermediate buffer
STILL OVERFLOWING
Flow Control
Flow Control

Reject at the boundary
CAF: Messaging Building Block

- CAF = C++ Actor Framework
- Implementation of the Actor Model
- Light-weight, type-safe, scalable
- Network transparency
Questions?

Docs: https://bro.github.io/broker
Chat: https://gitter.im/bro/broker
Code: https://github.com/bro/broker