libcppa Now:
High-Level Distributed Programming
Without Sacrificing Performance

Matthias Vallentin
matthias@bro.org

University of California, Berkeley

C++Now
May 14, 2013
Outline

1. Example Application: VAST

2. Designing Distributed Applications

3. Thinking libcppa
   • Interfacing with 3rd-party APIs
   • Agility Through Network Transparency
   • Behavior Composition
   • Monadic Composition
The Network Security Domain

Network Forensics & Incident Response

- Scenario: security breach discovered
- Analysts have to determine scope and impact

Analyst questions

- How did the attacker(s) get in?
- How long did they stay under the radar?
- What is the damage ($$$, reputation, data loss, etc.)?
- How to detect similar attacks in the future?

Challenges

- **Volume**: machine-generated data exceeds analysis capacities
- **Typing**: difficult to contextualize minimally typed data
- **Heterogeneity**: numerous log formats and processing styles
VAST: Visibility Across Space and Time
Architecture Overview

**VAST**

A **scalable, interactive** system to facilitate
- forensic analyses
- incident response

**Components**

- **Archive**: Compressed, serialized events
- **Index**: Encoded, compressed bitmaps
- **Segment/Partition**: Data scaling unit
- **Ingestion**: Sources: IDS, syslog, etc.
- **Query**: Sinks: Analyst, IDS, feed, etc.
VAST: Distributed Deployment

Analyst

VAST core

Diagram showing a network of interconnected components.
Outline

1. Example Application: VAST

2. Designing Distributed Applications

3. Thinking libcppa
   - Interfacing with 3rd-party APIs
   - Agility Through Network Transparency
   - Behavior Composition
   - Monadic Composition
Software Design in Distributed Applications
Challenge: Varying Deployment Scenarios

Machine 1

Component A

Component B

Machine 2

Component C
Challenge: Varying Deployment Scenarios

Machine 1

Component A

Component B

Machine 2

Component C

Pointer Passing

IPC & Serialization
Primitives for Programming Distributed Systems

Desired Building Blocks

- Flexible serialization
- Platform independence
- Network transparency
- Rich-typed messaging
- Asynchronous coding style
- Powerful concurrency model

C++ Reality

No existing light-weight middle-layer with high level of abstraction 😞

libcppa aims to fill that gap
Outline

1. Example Application: VAST

2. Designing Distributed Applications

3. Thinking libcppa
   - Interfacing with 3rd-party APIs
   - Agility Through Network Transparency
   - Behavior Composition
   - Monadic Composition
Outline

1. Example Application: VAST

2. Designing Distributed Applications

3. Thinking libcppa
   - Interfacing with 3rd-party APIs
   - Agility Through Network Transparency
   - Behavior Composition
   - Monadic Composition
Creating a TCP/IP Server

**VAST: Data Ingestion**

1. Accept new TCP/IP connection
2. Poll POSIX file descriptor of connection
3. Read from socket when data is available
4. Convert data into internal event format

**Existing solutions**
- Basic: `accept` & `fork` (or new thread)
- Boost Asio: non-blocking, but IoC
A Generic Asynchronous IPv4 Server

```cpp
template <typename Connection>
void server(uint16_t port, char const* addr, actor_ptr handler) {
  auto acceptor = network::ipv4_acceptor::create(port, addr);
  receive_loop( // libcoppa's blocking API
    on(atom("accept")) >> [&] {
      if (util::poll(acceptor->file_handle()) &&
          (auto io = acceptor->try_accept_connection())) {
        auto conn = spawn<Connection>((*io).first, (*io).second);
        send(handler, atom("connection"), conn);
      }
      self << last_dequeued();
    },
    on(atom("kill")) >> [] { self->quit(); }
  );
}
```
A Generic Asynchronous IPv4 Server

template <typename Connection>
void server(uint16_t port, char const* addr, actor_ptr handler) {
  auto acceptor = network::ipv4_acceptor::create(port, addr);
  receive_loop( // libcoppa's blocking API
    on(atom("accept")) >> [&] {
      if (util::poll(acceptor->file_handle()) &&
        (auto io = acceptor->try_accept_connection())) {
        auto conn = spawn<Connection>((*io).first, (*io).second);
        send(handler, atom("connection"), conn);
      }
      self << last_dequeued();
    },
    on(atom("kill")) >> [] { self->quit(); }
  );
}
A Generic Asynchronous IPv4 Server

template <typename Connection>
void server(uint16_t port, char const* addr, actor_ptr handler) {
  auto acceptor = network::ipv4_acceptor::create(port, addr);
  receive_loop( // libcoppa's blocking API
    on(atom("accept")) >> [&] {
      if (util::poll(acceptor->file_handle()) &&
          (auto io = acceptor->try_accept_connection())) {
        auto conn = spawn<Connection>((*io).first, (*io).second);
        send(handler, atom("connection"), conn);
        self << last_dequeued();
      },
      on(atom("kill")) >> [] { self->quit(); }
    );
}
template <typename Connection>
void server(uint16_t port, char const* addr, actor_ptr handler) {
  auto acceptor = network::ipv4_acceptor::create(port, addr);
  receive_loop(
      // libcppa's blocking API
      on(atom("accept")) >> [&] {
        if (util::poll(acceptor->file_handle())) {
          auto io = acceptor->try_accept_connection();
          auto conn = spawn<Connection>((*io).first, (*io).second);
          send(handler, atom("connection"), conn);
        }
        self << last_dequeued();
      },
      on(atom("kill")) >> [] { self->quit(); }
  );
}
template <typename Connection>
void server(uint16_t port, char const* addr, actor_ptr handler) {
  auto acceptor = network::ipv4_acceptor::create(port, addr);
  receive loop( // libcppa's blocking API
    on(atom("accept")) >> [&] {
      if (util::poll(acceptor->file_handle()) &&
          (auto io = acceptor->try_accept_connection())) {
        auto conn = spawn<Connection>((*io).first, (*io).second);
        send(handler, atom("connection"), conn);
      }
      self << last_dequeued();
    },
    on(atom("kill")) >> [] { self->quit(); }
  );
}

Behavior definition: start the accept loop or shut down the actor
template<typename Connection>
void server(uint16_t port, char const *addr, actor_ptr handler) {
    auto acceptor = network::ipv4_acceptor::create(port, addr);
    receive_loop(
        on(atom("accept")) >> [] {
            if (util::poll(acceptor->file_handle()) &&
                (auto io = acceptor->try_accept_connection())) {
                auto conn = spawn<Connection>((*io).first, (*io).second);
                send(handler, atom("connection"), conn);
            }
            self << last_dequeued();
        },
        on(atom("kill")) >> [] { self->quit(); }
    );
}
Outline

1. Example Application: VAST

2. Designing Distributed Applications

3. Thinking libc++
   - Interfacing with 3rd-party APIs
   - Agility Through Network Transparency
   - Behavior Composition
   - Monadic Composition
Developing Agile Systems

Provisioning

- Reality: skewed workload distributions
  - Requires *dynamic* provisioning
    - **Under**-provisioned: spin up actors
    - **Over**-provisioned: bring down actors

Agility

- Ability to handle/buffer peak loads
- Runtime re-configuration
  - without process restart
  - without losing in-memory state
class program {
public:
  void run() {
    auto host = config_.get("tracker.host");
    auto port = config_.as<unsigned>("tracker.port");
    if (config_.check("tracker-actor")) {
      tracker_ = spawn<id_tracker>("/path/to/id_file");
      publish(tracker_, port, host.data());
    } else {
      tracker_ = remote_actor(host, port);
    }
    ... // Further similar initializations.
  }
private:
  configuration config_;
  actor_ptr tracker_;
class program {
public:
  void run() {
    auto host = config_.get("tracker.host");
    auto port = config_.as<unsigned>("tracker.port");
    if (config_.check("tracker-actor")) {
      tracker_ = spawn<id_tracker>("/path/to/id_file");
      publish(tracker_, port, host.data());
    } else {
      tracker_ = remote_actor(host, port);
    }
    ... // Further similar initializations.
  }
private:
  configuration config_;
  actor_ptr tracker_;}

If actor should run in this process, 
spawn and publish it
class program {
public:
  void run() {
    auto host = config_.get("tracker.host");
    auto port = config_.as<unsigned>("tracker.port");
    if (config_.check("tracker-actor")) {
      tracker_ = spawn<id_tracker>("/path/to/id_file");
      publish(tracker_, port, host.data());
    } else {
      tracker_ = remote_actor(host, port);
    }
  }

private:
  configuration config_;
  actor_ptr tracker_;
class program {
public:
  void run() {
    auto host = config_.get("tracker.host");
    auto port = config_.as<unsigned>("tracker.port");
    if (config_.check("tracker-actor")) {
      tracker_ = spawn<id_tracker>("/path/to/id_file");
      publish(tracker_, port, host.data());
    } else {
      tracker_ = remote_actor(host, port);
    }
    ... // Further similar initializations.
  }
private:
  configuration config_;
  actor_ptr tracker_;
Outline

1. Example Application: VAST

2. Designing Distributed Applications

3. Thinking libcppa
   - Interfacing with 3rd-party APIs
   - Agility Through Network Transparency
   - Behavior Composition
   - Monadic Composition
Modularizing Code

Basic Techniques

▶ Straight-forward to reuse code in OO-style programs
▶ But how to reuse behavior in libcoppa?

Object-Oriented Inheritance

```cpp
default struct base {};
default struct derived : base {};
```

Object-Oriented Composition

```cpp
default struct foo {};
default struct bar { foo f };
```

Composing behavior in libcoppa

▶ Example:

```cpp
default partial_function f;
default partial_function g;
default partial_function h = f.or_else(g);

partial_function i;
default behavior j = h.or_else(i);
```
template <typename Derived>

class async_source : public event_based_actor {
public:
  async_source(actor_ptr upstream) {
    running_ = (
      on_arg_match >> [=](event const& e) { /* work */ },
      on_arg_match >> [=](std::vector<event> const& v) { /* work */ }
    );
  }

  void init() override {
    become(running_.or_else(static_cast<Derived*>(this)->impl));
  }

private:
  partial_function running_; 
};

struct file_source : async_source<file_source> { behavior impl; };
Example: Base Class for Event Sources (simplified)

```cpp
template <typename Derived>
class async_source : public event_based_actor {
public:
  async_source(actor_ptr upstream) {
    running_ = (
      on_arg_match >> [=](event const & e) {
        /* work */
      },
      on_arg_match >> [=](std::vector<event> const & v) {
        /* work */
      });
  }

void init() override {
  become(running_.or_else(
    static_cast<Derived*>(this)->impl));
}
private:
  partial_function running_;}

struct file_source : async_source<file_source> { behavior impl; }
```

*libcppa's event-based actors require implementation of init()*. 
template <typename Derived>
class async_source : public event_based_actor {
public:
  async_source(actor_ptr upstream) {
    running_ = (
      on_arg_match >> [=](event const& e) { /* work */ },
      on_arg_match >> [=](std::vector<event> const& v) { /* work */ }
    );
  }
  void init() override {
    become(running_.or_else(static_cast<Derived*>(this)->impl));
  }
private:
  partial_function running_;}

struct file_source : async_source<file_source> { behavior impl; };
Example: Base Class for Event Sources (simplified)

```cpp
template <typename Derived>
class async_source : public event_based_actor {
public:
  async_source(actor_ptr upstream) {
    running_ = (  
      on_arg_match >> [=](event const& e) { /* work */ },  
      on_arg_match >> [=](std::vector<event> const& v) { /* work */ }  
    );
  }
  void init() override {
    become(running_.or_else(static_cast<Derived*>(this)->impl));
  }
private:
  partial_function running_;  
};

struct file_source : async_source<file_source> { behavior impl;  
};
```

Composition of partial functions (or behavior)
Outline

1. Example Application: VAST

2. Designing Distributed Applications

3. Thinking libcxx
   - Interfacing with 3rd-party APIs
   - Agility Through Network Transparency
   - Behavior Composition
   - Monadic Composition
Developing Message Protocols

Asynchronous Messaging

- Decoupled send and receive
  1. Pattern-match dequeued message
  2. Invoke corresponding handler
  3. Send message (optional)
- libcppa: send(..)
- Loose coupling

Synchronous Messaging

- Lockstep: matching send and receive
  1. Send a message
  2. Push new behavior to handle reply
  3. Pop behavior
- libcppa: sync_send(..)
- How to compose?
The Lack of Monads in C++

Composition

- C++ lacks a powerful language primitive for composition: `monads`
- `(>>=) :: (Monad m) => m a -> (a -> m b) -> m b`

HasC++all

```cpp
sync_send(b, atom("ping"))
  >>= on(atom("pong")) >> [=] { return sync_send(b, atom("stop")); }
  >>= on(atom("done"), arg_match) >> [=](int i) { f(i); }
```

libc++

```cpp
sync_send(b, atom("ping")).then(
  on(atom("pong")) >> [=] { sync_send(b, atom("stop")).then(
    on(atom("done"), arg_match) >> [=](int i) { f(i); });
  });
```
Summary

Experience using libcppa
- Offered abstractions facilitate solving domain challenges
- Asynchronous coding style with local reasoning (no IoC)
- Easy integration with 3rd-party APIs
- Flexible deployment scenarios support highly dynamic systems
- Low overhead (1–3% CPU time)

Future Work
- Improve runtime type debugging and diagnostics

Parting thoughts
- Actor model as natural paradigm to address today’s challenges
  - libprocess, jss::actor, Casablanca, ...
- C++1? wish list:
  - Monads → composition of asynchronous operations
  - Pattern matching → type-safe dispatching
Thank You...Questions?

FIN

https://github.com/Neverlord/libcppa
https://github.com/mavam/vast

IRC at Freenode: #libcppa, #vast