CAF – A Short Introduction

Dominik Charousset
dominik.charousset@haw-hamburg.de

iNET RG, Department of Computer Science
Hamburg University of Applied Sciences

October 2014

Hochschule für Angewandte
Wissenschaften Hamburg
Hamburg University of Applied Sciences

The Problem With Implicit Sharing

When writing concurrent programs:

- Stateful objects need to be synchronized (if shared)
- Developer is responsible for thread-safety
- Challenges are ...
  - Race conditions (“solved” by locks)
  - Deadlocks/Lifelocks (caused by locks)
  - Poor scalability due to queueing (Coarse-Grained Locking)
  - Very high complexity (Fine-Grained Locking)
- Time-dependent errors make testing (almost) impossible

$\Rightarrow$ Expert knowledge & experience required

The Problem With Implicit Sharing

When writing concurrent programs:

- Stateful objects need to be synchronized (if shared)
- Developer is responsible for thread-safety
- Challenges are ...
  - Race conditions (“solved” by locks)
  - Deadlocks/Lifelocks (caused by locks)
  - Poor scalability due to queueing (Coarse-Grained Locking)
  - Very high complexity (Fine-Grained Locking)
- Time-dependent errors make testing (almost) impossible

$\Rightarrow$ Expert knowledge & experience required

Compose Synchronized Classes

```
class Subject {
 public:
  void subscribe(function<void(int)> fun) {
    unique_lock<mutex> guard{m_mtx};
    m_subscribers.push_back(move(fun));
  }
  void broadcast(int value) {
    unique_lock<mutex> guard{m_mtx};
    for (auto& s : m_subscribers) s(value);
  }
 private:
  mutex m_mtx;
  vector<function<void(int)>> m_subscribers;
};
```

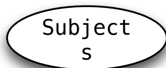
Compose Synchronized Classes

```
class FooBar {
 public:
  void foo(Subject* s) {
    unique_lock<mutex> guard{m_mtx};
    m_subjects.push_back(s);
    s->subscribe([=](int v) {
      /*...*/ bar(v); /*...*/
    });
    // ...
  }
  void bar(int value) {
    unique_lock<mutex> guard{m_mtx};
    // ...
  }
 private:
  vector<Subject*> m_subjects;
  mutex m_mtx;
};
```

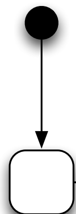
Compose Synchronized Classes

Thread1

Thread2



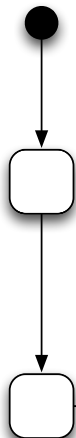
Compose Synchronized Classes

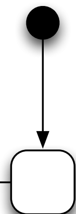


Compose Synchronized Classes

Thread1

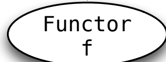
Thread2



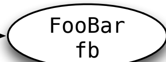


broadcast(42)

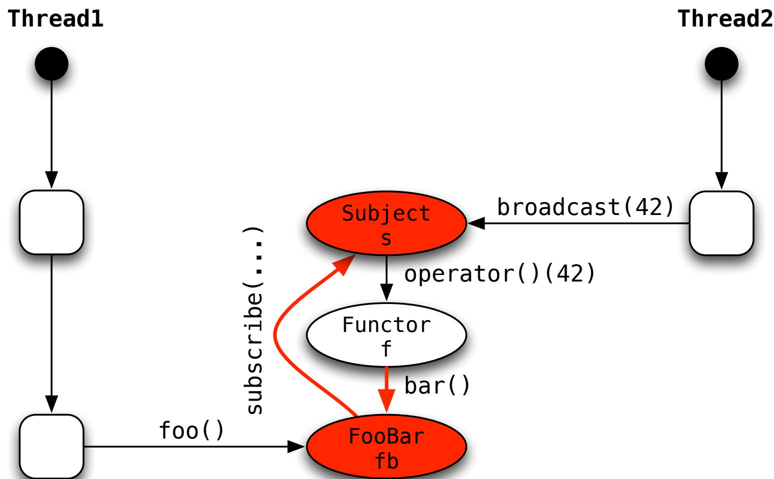
operator()(42)



foo()



Compose Synchronized Classes



Locks Are Not Composable

"Mutable, stateful objects are the new spaghetti code."
– Rich Hickey

- Libraries with threads & locks are no longer black boxes
- Composition of two thread-safe classes can deadlock
- User has to know about implementation details:
  - Which code runs asynchronously/where?
  - Which functions are "thread-safe"?
  - Which function uses which lock?

⇒ Abstraction of OO programming unfolds

Locks Are Not Composable

"Mutable, stateful objects are the new spaghetti code."
– Rich Hickey

- Libraries with threads & locks are no longer black boxes
- Composition of two thread-safe classes can deadlock
- User has to know about implementation details:
  - Which code runs asynchronously/where?
  - Which functions are "thread-safe"?
  - Which function uses which lock?

⇒ Abstraction of OO programming unfolds

Locks Are Not Composable

"Mutable, stateful objects are the new spaghetti code."
– Rich Hickey

- Libraries with threads & locks are no longer black boxes
- Composition of two thread-safe classes can deadlock
- User has to know about implementation details:
  - Which code runs asynchronously/where?
  - Which functions are "thread-safe"?
  - Which function uses which lock?

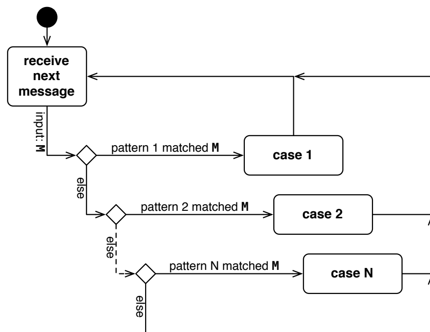
⇒ Abstraction of OO programming unfolds

The Actor Model

Actors are concurrent entities, that ...

- Communicate via message passing
- Do not share state
- Can create (“spawn”) more actors
- Can monitor other actors

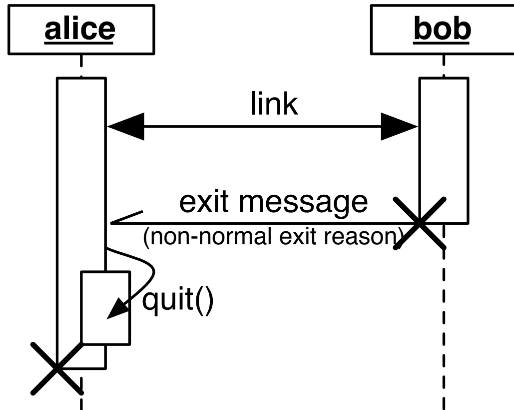
The Actor Model – Programming Model



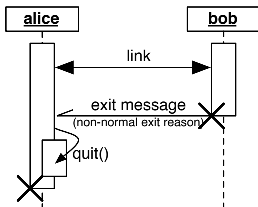
Actor Programming *is* Message-Oriented Programming

- Actors are *active* objects
- No direct method invocation, only messages
- Messages passing *hides* location of receiver
- Receiver pattern matches on content of incoming messages

The Actor Model – Linking of Actors



The Actor Model – Linking of Actors



- Actors can *link* their lifetime
- Errors are propagated through exit messages
- When receiving an exit message:
  - Actors fail for the same reason per default
  - Actors can *trap* exit messages to handle failure manually
- Build systems where all actors are alive or have collectively failed

The Actor Model – Linking of Actors

Trapping exit messages enables:

- Notification on success (normal exit reason)
- Report errors back to client (non-normal exit reason)
- Re-deployment of workers on (hardware) node failure
- *Supervising* spawned workers

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction for developing software
- Abstraction over deployment
  - Flexible & modular systems
  - Managing heterogeneous environments (but not yet on HW level)
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction for developing software
- Abstraction over deployment
  - Flexible & modular systems
  - Managing heterogeneous environments (but not yet on HW level)
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction for developing software
- Abstraction over deployment
  - Flexible & modular systems
  - Managing heterogeneous environments (but not yet on HW level)
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction for developing software
- Abstraction over deployment
  - Flexible & modular systems
  - Managing heterogeneous environments (but not yet on HW level)
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

CAF – Actors in C++11

- CAF is an actor system based on C++11
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW
  - Multi-core systems
- Uses internal DSL for pattern matching of messages

CAF – Actors in C++11

- CAF is an actor system based on C++11
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW
  - Multi-core systems
- Uses internal DSL for pattern matching of messages

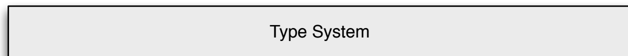
CAF – Actors in C++11

- CAF is an actor system based on C++11
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW
  - Multi-core systems
- Uses internal DSL for pattern matching of messages

CAF – Actors in C++11

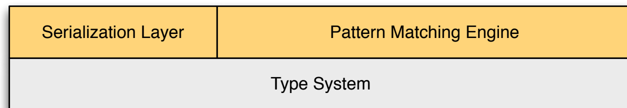
- CAF is an actor system based on C++11
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW
  - Multi-core systems
- Uses internal DSL for pattern matching of messages

CAF Core Architecture



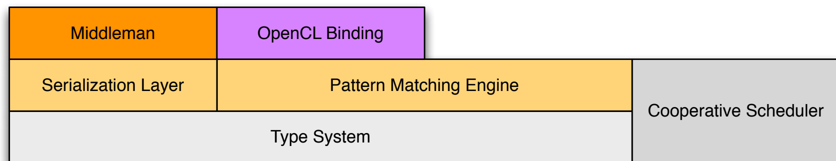
1/8

CAF Core Architecture



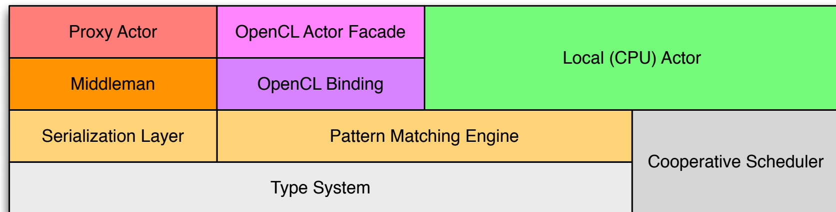
2/8

CAF Core Architecture



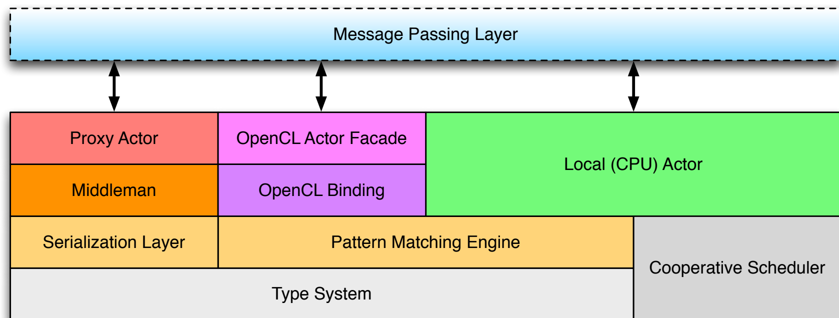
3/8

CAF Core Architecture



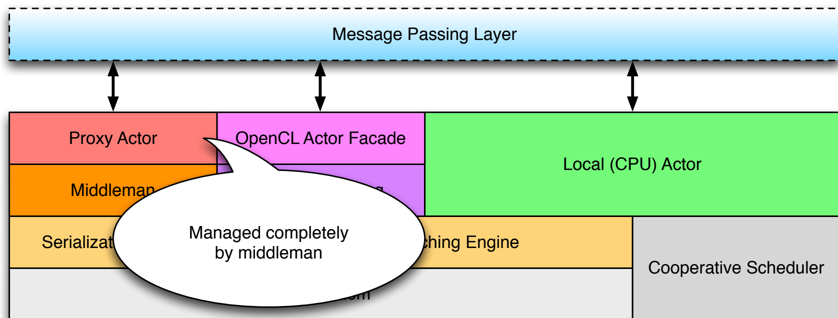
4/8

CAF Core Architecture



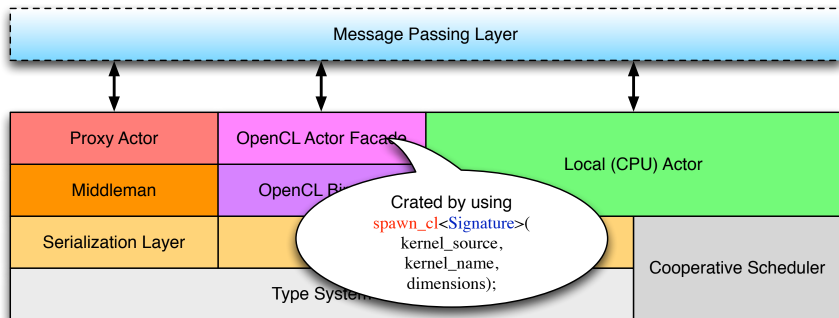
5/8

CAF Core Architecture



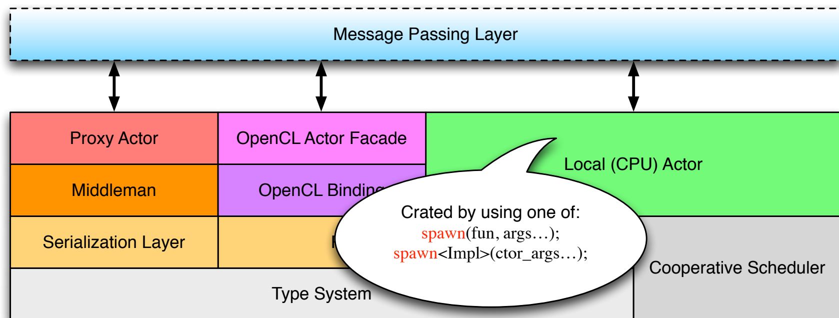
6/8

CAF Core Architecture



7/8

CAF Core Architecture



Classes vs. Actors

```
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

```
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

Classes vs. Actors

```
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

```
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

Classes vs. Actors

```
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

```
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

Classes vs. Actors

```
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

```
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

API – Creating Actors

```
// args: constructor arguments for Impl
template<class Impl,
         spawn_options Os = no_spawn_options,
         typename... Ts>
actor spawn(Ts&&... args);

// args: functor followed by its arguments
template<spawn_options Os = no_spawn_options,
         typename... Ts>
actor spawn(Ts&&... args);
```

- Create actors from either functors or classes
- Spawn options can be used for monitoring, detaching, etc.
- Creates event-based actors per default

API – Event-based Actor Class

```
class event_based_actor : ... {

  template<typename... Ts>
  void send(actor whom, Ts&&... what);

  template<typename... Ts>
  response_handle sync_send(actor whom, Ts&&... what);

  void become(behavior bhvr);

  void quit(uint32_t reason);

  // ...

};
```

- Base for class-based actors
- Type of implicit `self` pointer for functor-based actors

API – Remote Communication

```
// makes actor accessible via network
void publish(actor whom, uint16_t port);

// get handle to remotely running actor
actor remote_actor(std::string host, uint16_t port);
```

- Message passing is network transparent
- Both local and remote actors use handles of type `actor`
- Network primitives not exposed to programmer

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
void math_client(event_based_actor* self, actor ms) {
  sync_send(ms, 40, 2).then(
    [=](int result){
      cout << "40 + 2 = " << result << endl;
      }
  );
}
// spawn(math_client, spawn(math_server));
```

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
voi                                  self, actor ms) {
  s
    [](int result){
      cout << "40 + 2 = " << result << endl;
      }
  );
}
// spawn(math_client, spawn(math_server));
```

return message handler for incoming messages (used until replaced or actor is done)

Example

```
behavior math_server() {
  r
  };
}
void math_client(event_based_actor* self, actor ms) {
  sync_send(ms, 40, 2).then(
    [=](int result){
      cout << "40 + 2 = " << result << endl;
      }
  );
}
// spawn(math_client, spawn(math_server));
```

send a message and then
wait for response
(using a "one-shot handler")

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  }
}
vo                                  * self, actor ms) {
  s
    [=](int result){
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// spawn(math_client, spawn(math_server));
```

this actor "loops" forever
(or until it is forced to quit)

Example

```
behavior math_server() {
  r
  };
}
void math_client(event_based_actor* self, actor ms) {
  sync_send(ms, 40, 2).then(
    [=](int result){
      cout << "40 + 2 = " << result << endl;
      }
  );
}
// spawn(math_client, spawn(math_server));
```

this actor sends one message and receives one messages

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
voi                              r* self, actor ms) {
  s
    cout <<  40 + 2 =  << result << endl;
    }
  );
}
// spawn(math_client, spawn(math_server));
```

spawn server & client

Serialization in CAF

- Non-intrusive serialization backend
- User-defined types need to be *announced*
- POD-like data: pointer to members or getter + setter
- Complex data: implementation of custom `uniform_type_info`
- All announced types can use `caf::to_string`

Serialization in CAF – PODs

```
struct foo {
  std::vector<int> a;
  int b;
};
// required by announce()
bool operator==(const foo& lhs, const foo& rhs) {
  return  lhs.a == rhs.a
       && lhs.b == rhs.b;
}
int main(int, char**) {
  announce<foo>(&foo::a, &foo::b);
  // ...
}
```

Patterns for Actor Programming

- Spawn one actor per task, keep individual actors simple
- Compose complex behavior out of small, easily testable actors
- Push stateful operations to new actors
- Use "recursive" message loops (no stack overflow possible)
- Do not block indefinitely, define "continuation points"

Patterns for Actor Programming

- Spawn one actor per task, keep individual actors simple
- Compose complex behavior out of small, easily testable actors
- Push stateful operations to new actors
- Use "recursive" message loops (no stack overflow possible)
- Do not block indefinitely, define "continuation points"

Patterns for Actor Programming

- Spawn one actor per task, keep individual actors simple
- Compose complex behavior out of small, easily testable actors
- Push stateful operations to new actors
- Use "recursive" message loops (no stack overflow possible)
- Do not block indefinitely, define "continuation points"

Patterns for Actor Programming

- Spawn one actor per task, keep individual actors simple
- Compose complex behavior out of small, easily testable actors
- Push stateful operations to new actors
- Use “recursive” message loops (no stack overflow possible)
- Do not block indefinitely, define “continuation points”

Patterns for Actor Programming

- Spawn one actor per task, keep individual actors simple
- Compose complex behavior out of small, easily testable actors
- Push stateful operations to new actors
- Use "recursive" message loops (no stack overflow possible)
- Do not block indefinitely, define "continuation points"

Interruptible Computation

```
// adds numbers [i, last) unless 'stop' is received
void counter(event_based_actor* self, actor client,
             size_t value, size_t i, size_t last) {
  if (i == last) {
    self->send(client, value);
    self->quit();
    return;
  }
  self->become(
    on(atom("stop")) >> [=] {
      self->quit();
    },
    after(std::chrono::seconds(0)) >> [=] {
      counter(self, client, value + i, i + 1, last);
    }
  );
}
```

Thank you for your attention!

Home page: http://actor-framework.org

Sources: https://github.com/actor-framework/actor-framework