CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications

Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt
{dominik.charousset,raphael.hiesgen,t.schmidt}@haw-hamburg.de

Dept. Computer Science
Hamburg University of Applied Sciences
Germany

October 2014, AGERE!@SPLASH
Previous Work

- Implemented native actor library `libcppa` actor library in C++
  - Target at both high-performance and embedded environments
  - Allow millions of lightweight actors

- Extended the actor model with publish/subscribe semantics
  - Original actor model only foresees 1:1 communication
  - Internet scale requires loose coupling

- Support heterogeneous hardware components
  - GPUs can outperform CPUs by orders of magnitude
  - Transparent integration of OpenCL allows flexible deployment
Rebranding & Modularization

Our approach to a growing userbase with diverse requirements:

- Move from a monolithic library to an open framework
- Split functionality into (optional) modules
- Enable customization via extensible framework structure
- Central project homepage\(^1\) linking to all activities

\(^1\)http://actor-framework.org
Agenda

1. Type-safe Message Passing
2. Scheduling Infrastructure
3. Runtime Inspection & Debugging
4. Conclusion & Outlook
Agenda

1. Type-safe Message Passing
2. Scheduling Infrastructure
3. Runtime Inspection & Debugging
4. Conclusion & Outlook
Problem of Dynamic Typing

The original model\(^2\) defines actors in terms of
- (Untyped) message passing primitives
- Pattern matching

⇒ Extensive integration testing required
- Coding errors occur at runtime
- Non-local dependencies are hard to track manually

Type-safe Message Passing

Lift type system of C++ and make it applicable to actor interfaces

- Compiler statically checks protocols between actors
- Protocol violation cannot occur at runtime
- Compiler verifies both incoming and outgoing messages:

```cpp
using math =
  typed_actor<
    replies_to<int, int>::with<int>,
    replies_to<float>::with<float, float>>;
// ...
auto ms = typed_spawn(...);
sync_send(ms, 10, 20).then(
  [](float result) {
    // compiler error: result is int, not float
  }
);
```
Agenda

1. Type-safe Message Passing
2. Scheduling Infrastructure
3. Runtime Inspection & Debugging
4. Conclusion & Outlook
Scalability of Scheduling

CAF aims at scaling to millions of actors on hundreds of processors

- Actors cannot be implemented (efficiently) as threads
- Running in userspace prohibits preemption
- Classical thread pool or centralized scheduler has limitations
  - Central job queue is a bottleneck per se
  - Short-lived tasks cause significant runtime overhead
  - *Could* schedule actors for real-time with a priori knowledge

---

Centralized Scheduling Issue

Divide & conquer: $2^{20}$ actors with libcoppa (central scheduling, 2013)

```
libcppa reached maximum performance on 8 cores for divide & conquer algorithms
```
Scheduling Approaches

- **Active dispatching**
  - Central task management
  - One (or more) threads manage others
  - High communication overhead

- **Shared work queues**
  - Reactive task management
  - Workers access one (or more) shared queues
  - Frequent access to shared data is a likely performance bottleneck

- **Individual work queues**
  - Decentralized, reactive task management
  - Workers communicate only when idle
  - Minimizes synchronizations between threads
Work Stealing

Decentralized scheduling using Work Stealing\(^4\)

- One job queue and worker per core
- Worker tries *stealing* work items from others when idle
- Stealing is a rare event for most work loads\(^5\)
- *But:* A priori knowledge cannot be exploited (no global view)

---


Work Stealing

Queue 1

Job 1

Job 2

Job 3

Victim
Worker 1

Queue 2

Job 3

Thief
Worker 2

Queue P

Job N

Worker P

Steal
Configurable Scheduling in CAF

Framework has no a priori knowledge → Work Stealing as default

- Using Work Stealing, CAF scales up to at least 64 cores
- Developers can deploy custom scheduler using

```cpp
template <class Policy = work_stealing>
void set_scheduler(size_t num_workers = ..., size_t max_msgs = indefinite);
```

- `max_msgs` restricts # of messages actors can consume at once
  - Low value increases fairness and avoids bursts
  - High value minimizes queue access, usually maximizing throughput
- `Policy` can be implemented to exploit a priori knowledge, if possible
Scheduling Infrastructure

Divide & conquer: $2^{20}$ actors with CAF

Dominik Charousset
iNET – HAW Hamburg
Scheduling Infrastructure

Mixed operations under work load with CAF

![Graph showing the time in seconds (y-axis) against the number of cores (x-axis) for different frameworks. Legend includes ActorFoundry, CAF, Charm, Erlang, and Scala.]
Agenda

1. Type-safe Message Passing
2. Scheduling Infrastructure
3. Runtime Inspection & Debugging
4. Conclusion & Outlook
Debugging of distributed systems is inherently complex
- Non-trivial program flow
- No global clock
- Diverging states

Recording messages crucial for on-line or post-mortem debugging

Erroneous behavior can be reproduced using message replaying

Visualization tools can help understanding complex errors

---


.Runtime Inspection & Debugging

Node A
actor A
actor B
 ...
Node N
actor C
actor D
 ...
Nexus
Frontend (e.g. shell)

Dominik Charousset
iNET – HAW Hamburg
Runtime Inspection & Debugging

Nexus
Frontend (e.g. shell)
Node A
P1
...
...
Node N
PN
actor A
actor B
actor C
actor D
actor
Probes

- Intercept & forward three kinds of messages to the Nexus:
  - **Activity events**: incoming & outgoing messages
  - **Error events**: network & system failures
  - **Runtime statistics**: periodic collection of CPU load, etc.
The Nexus

- Provides global view of the distributed system
- Receives & collects events from Probes
- Statefully configures verbosity of Probes
Runtime Inspection & Debugging

Frontend application categories

- **Observing agents**: monitoring & threshold-based alerts
- **Supervising agents**: active manipulation of running app.
- **Monitoring & visualization**: access to aggregate state
  ⇒ For instance, an *interactive inspection shell*

---

Dominik Charousset
iNET – HAW Hamburg
Interactive Inspection Shell

- Allows users to inspect distributed system
- In global mode:
  - Global view to the system
  - Access to individual participating nodes
- In node mode:
  - Access to statistics such as RAM usage, CPU load, etc.
  - Direct interaction with actors on that node
Agenda

1. Type-safe Message Passing
2. Scheduling Infrastructure
3. Runtime Inspection & Debugging
4. Conclusion & Outlook
Conclusion

- CAF is a robust, scalable platform for native actor programming
- Strong emphasis on low mem. footprint and performance
- Type-safe messaging interfaces
- Open scheduling infrastructure with efficient default
- First step towards debugging distributed actors
Outlook

- Scale down to IoT devices (port CAF to RIOT-OS\(^8\))
- Load balancing for massively parallel, distributed systems
- Monitoring and debugging tools based on current platform
- Robust security layer for the IoT: subsuming strong authentication of actors in combination with opportunistic encryption

\(^8\)http://riot-os.org
Thank you for your attention!

Homepage:  http://actor-framework.org

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

iNET Working Group:  http://inet.cpt.haw-hamburg.de
References

Carl Hewitt, Peter Bishop, and Richard Steiger.
A Universal Modular ACTOR Formalism for Artificial Intelligence.

M.L. Dertouzos and A.K. Mok.
Multiprocessor Online Scheduling of Hard-Real-Time Tasks.

Robert D. Blumofe and Charles E. Leiserson.
Scheduling Multithreaded Computations by Work Stealing.

Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu.
Work-stealing Without the Baggage.

Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica.
Replay debugging for distributed applications.

Terry Stanley, Tyler Close, and Mark S Miller.
Causeway: A message-oriented distributed debugger.