Fast and Flexible Framework for Simulation of Distributed Systems

Oleg Sukhoroslov, Artem Makogon

Kharkevich Institute for Information Transmission Problems

HSE UNIVERSITY

Distributed Systems

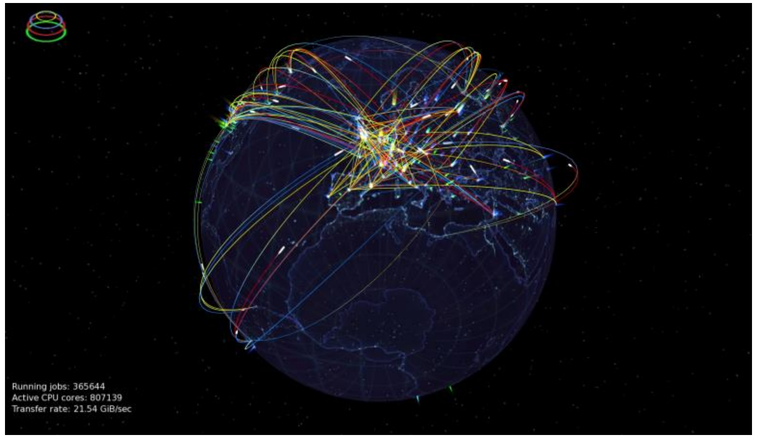
Modern systems and applications are increasingly distributed

- High performance, scalability, availability, decentralization, flexibility…

Distributed systems are hard to build, test and operate

- Asynchrony, concurrency, absence of global clock, partial failures, heterogeneity, dynamicity, scale…

How to solve related problems, design and optimize systems, and train new specialists?



Challenges

Researchers: How to evaluate proposed method? How to compare and becnhmark alternative methods? How to reproduce results from a paper?

Practitioners: How to evaluate design of a new system or alternative designs? How does this change improve the operation of existing system? What if …?

Educators: How to expose students to problems that occur in modern systems?

- Analytical models are not sufficient
- Small lab environment has limited capabilities
- Building a (copy of) real-scale system is too expensive
- Results of “in vivo” experiments are not reproducible
- Running experiments on working system is dangerous

Simulation

The studied system is replaced by a computer model that imitates the real system (components, processes) with sufficient accuracy

- Real system is not needed
- Inexpensive, moderate resource requirements
- Faster experiments, no real-time
- Full control over environment and reproducibility
- Any system configuration or scenario
- Virtual environment for education purposes

Build simulator from scratch or use an existing solution?

Existing Solutions

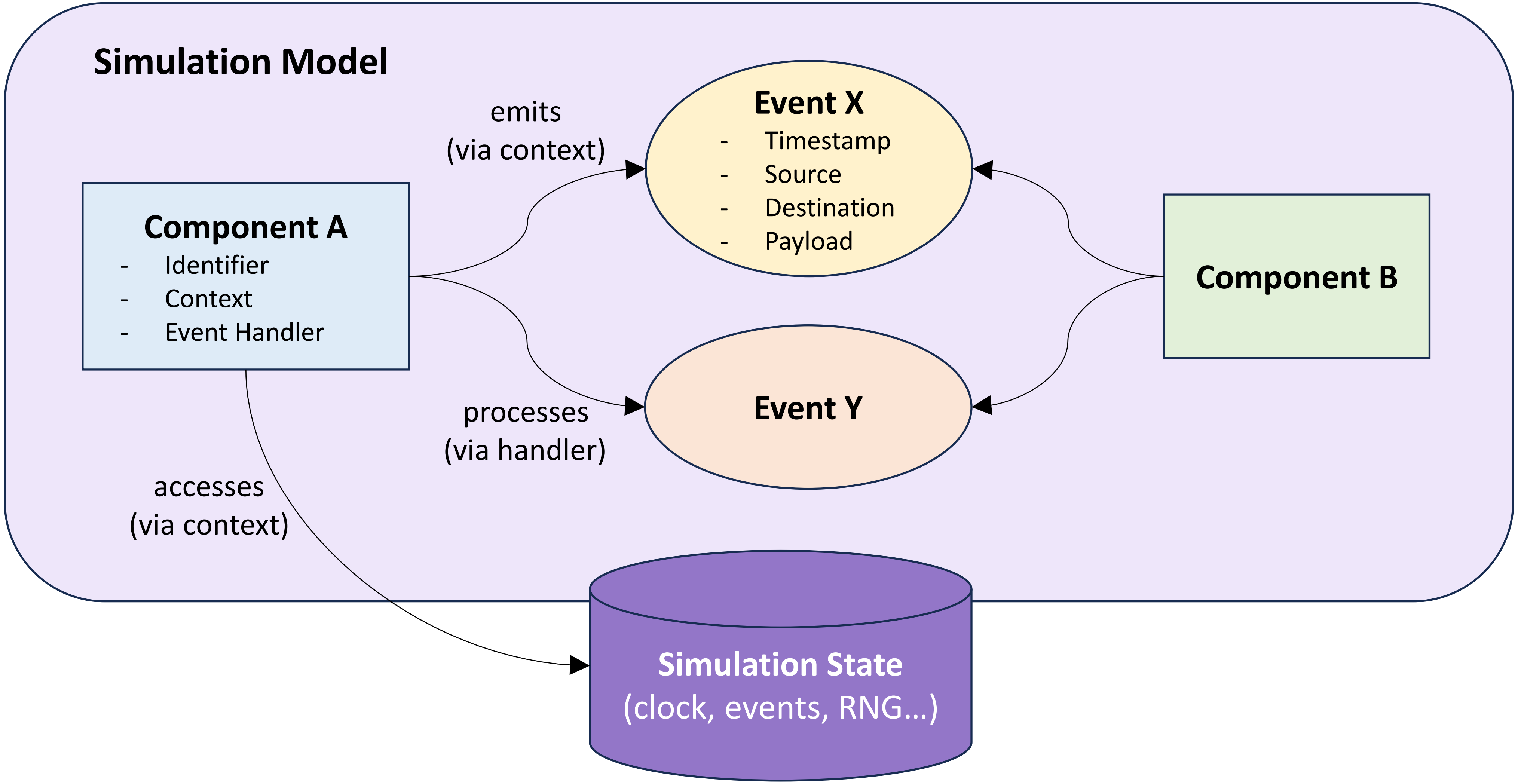
- Simulators tailored for specific application and research domains
  - Mostly built by researchers for their projects (and often abandoned later)
  - Limited reusability, extensibility and support
- General-purpose simulation frameworks and platforms
  - Provide necessary components to develop simulators for different use cases
  - Examples: SimGrid, CloudSim, OpenDC
  - Still tailored to specific domain (HPC, cloud, data center)
  - Lack of convenient and flexible general-purpose programming models
  - Performance is not sufficient for large-scale simulations

Domain-agnostic and high-performance simulation framework
with flexible and expressive programming model?

SimCore

- Discrete-event simulation framework
- Generic event-driven programming model
  - Any domain, even beyond distributed systems
- Callback-based and asynchronous programming
  - Any execution logic
- No domain-specific abstractions and primitives
  - Provided via separate libraries
- Implemented in Rust language
  - Performance, resource efficiency, memory safety
- Used to build several domain-specific simulators
  - Two are presented on this conference

Basic Concepts



Simulation Interface

```
fn main() {
    // Create simulation with specified random seed
    let mut sim = Simulation::new(123);

    // Create and register components
    let proc1 = Process::new(0.1, sim.create_context("proc1"));
    let proc1_ref = Rc::new(RefCell::new(proc1));
    sim.add_handler("proc1", proc1_ref.clone());
    let proc2 = Process::new(0.1, sim.create_context("proc2"));
    let proc2_ref = Rc::new(RefCell::new(proc2));
    let proc2_id = sim.add_handler("proc2", proc2_ref);

    // Ask proc1 to send request to proc2
    proc1_ref.borrow().send_request(proc2_id);

    // Run simulation until there are no pending events
    sim.step_until_no_events();
    println!("Simulation time: {:.2}", sim.time());
}
```

Events and Component Definition

```rust
#[derive(Clone, Serialize)]
struct Request {
    time: f64,
}

#[derive(Clone, Serialize)]
struct Response {
    req_time: f64,
}
```

```rust
struct Process {
    net_delay: f64,
    ctx: SimulationContext,
}

impl Process {
    pub fn new(net_delay: f64, ctx: SimulationContext) -> Self {
        Self { net_delay, ctx }
    }

    fn send_request(&self, dst: Id) {
        self.ctx.emit(Request { time: self.ctx.time() }, dst, self.net_delay);
    }

    fn on_request(&self, src: Id, req_time: f64) {
        let proc_delay = self.ctx.gen_range(0.5..1.0);
        self.ctx.emit(Response { req_time }, src, proc_delay + self.net_delay);
    }

    fn on_response(&self, req_time: f64) {
        let response_time = self.ctx.time() - req_time;
        println!("Response time: {:.2}", response_time);
    }
}
```

Receiving Events via Callbacks

```
impl EventHandler for Process {
    fn on(&mut self, event: Event) {
        cast!(match event.data {
            Request { time } => {
                self.on_request(event.src, time)
            }
            Response { req_time } => {
                self.on_response(req_time)
            }
        })
    }
}
```

- Works well for simple cases by organizing all event processing logic in EventHandler
- Complicates implementation of **multi-step activities** (steps are spread across multiple functions)

Async Mode

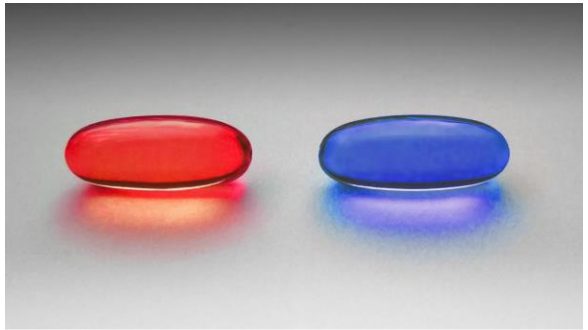
```
impl Process { ...
    fn send_request(self: Rc<Self>, dst: Id) {
        self.ctx.spawn(self.clone().send_request_and_get_response(dst))
    }

    async fn send_request_and_get_response(self: Rc<Self>, dst: Id) {
        let send_time = self.ctx.time();
        self.ctx.emit(Request {}, dst, self.net_delay);
        self.ctx.recv_event::<Response>().await;
        let response_time = self.ctx.time() - send_time;
        println!("Response time: {:.2}", response_time);
    }

    async fn process_request(self: Rc<Self>, src: Id) {
        self.ctx.sleep(self.ctx.gen_range(0.5..1.0)).await;
        self.ctx.emit(Response {}, src, self.net_delay);
    }
}
```

```
impl StaticEventHandler for Process {
    fn on(self: Rc<Self>, event: Event) {
        cast!(match event.data {
            Request {} => { self.ctx.spawn(self.clone().process_request(event.src)) }
        })
    }
}
```

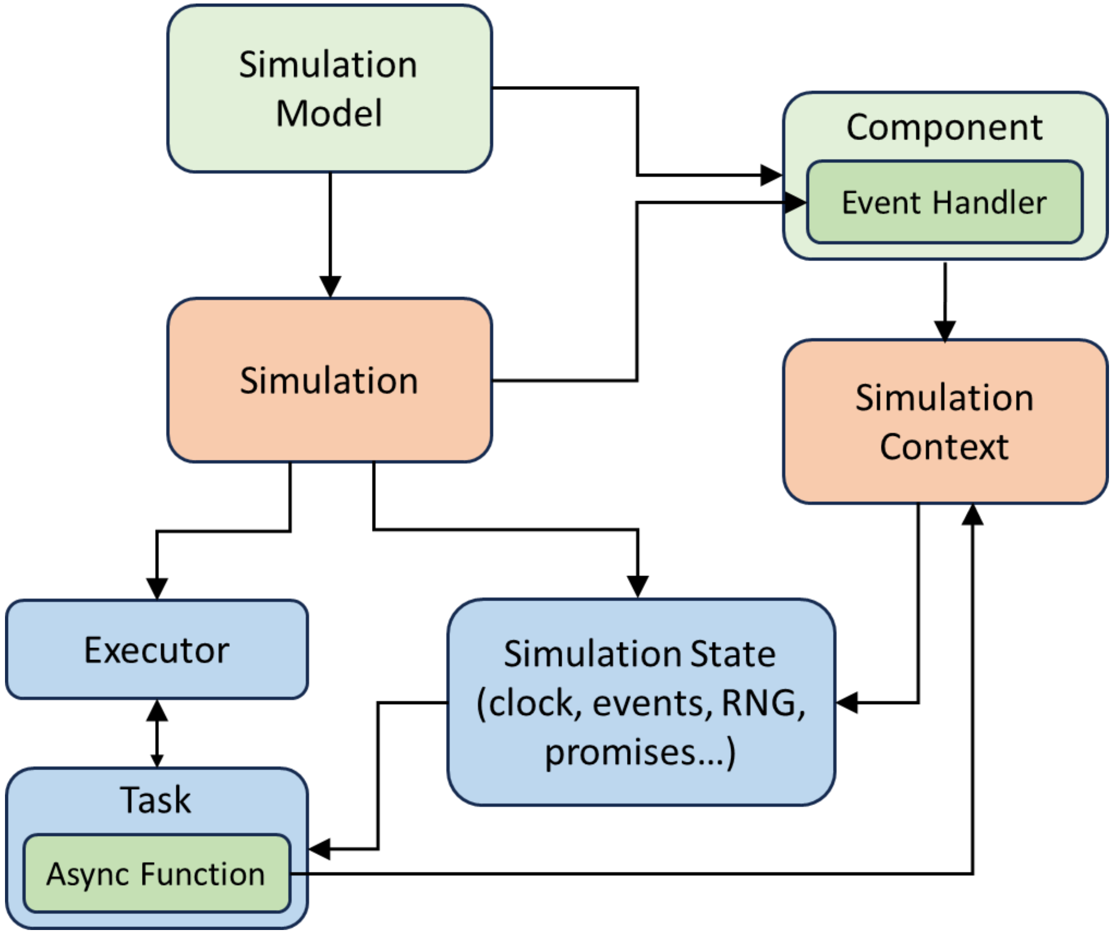
Combining Advantages of Both Approaches



- Callbacks (EventHandler)
  - Describing simple event processing logic
  - Receiving events triggering a complex logic
- Async mode
  - Describing complex logic with multiple steps and waiting
  - Waiting for multiple events simultaneously using *join* and *select* primitives
  - Selective receive of events by a user-defined key

Implementation

- ~2000 lines of code in Rust
- Hybrid event storage
  - Priority queue by default
  - Deque for ordered events
- Async programming support from Rust standard library and compiler, primitives from *futures* crate
- Simulation execution is performed sequentially using a *single thread*



Use Cases

Common libraries: reusable primitives for building simulations

- Models of compute and storage resources, network and power consumption
- Generic model of sharing resource with limited throughput

Domain-specific libraries: complete simulation solutions

- DSLab DAG (scheduling of computations represented as directed acyclic graphs)
- DSLab IaaS (resource management in Infrastructure-as-a-Service clouds)
- DSLab FaaS (resource management in Function-as-a-Service clouds)
- AnySystem (deterministic simulation and testing of distributed systems)
- ClusterSim (modeling of cluster computing workloads and scheduling problems)
- BOSS (simulation of BOINC volunteer computing platform)

Performance Evaluation: Ping-Pong

- *N* processes communicate with *P* peers by exchanging *Ping/Pong* messages
- Message transmission time is modeled using a fixed delay
- Allows to evaluate raw performance of simulation framework (almost no user code)
- SimCore is 20-40 times faster than SimGrid, can process up to 13M events/second

			Execution Time, seconds		
Processes	Peers	Iterations	SimCore (callback)	SimCore (async)	SimGrid
2	1	1000000	0.31	0.63	10.15
2	1	10000000	2.89	6.18	101
1000	10	1000	0.35	0.51	6.99
10000	100	1000	4.22	6.25	152
100000	100	1000	57	106	2439
1000000	100	1000	837	1306	39657

Performance Evaluation: Master-Workers

- Simulates a heterogeneous distributed computing system processing *T* tasks
- Tasks are dynamically distributed among *W* worker nodes (scheduling takes noticeable time)
- Uses common resource models: compute, storage, network (without or with bandwidth sharing)
- SimCore allows to simulate a system with 1M nodes in several minutes while using 8 GB of RAM

			Execution (Task Scheduling) Time, seconds		
Workers	Tasks	B/w sharing	SimCore (callback)	SimCore(async)	SimGrid
100	10000	No	0.10 (0.05)	0.12 (0.05)	0.51 (0.03)
1000	10000	No	0.09 (0.02)	0.10 (0.02)	4.89 (0.22)
1000	100000	No	0.84 (0.30)	0.98 (0.30)	26.97 (4.76)
10000	1000000	No	14.09 (4.84)	16.90 (4.95)	7630 (780)
100000	1000000	No	18.81 (2.72)	22.14 (3.13)	-
1000000	1000000	No	30.05 (2.59)	34.51 (2.58)	out of memory
1000000	10000000	No	275 (36.43)	326 (40.88)	out of memory
100	10000	Yes	0.13 (0.06)	0.14 (0.06)	0.51 (0.03)
1000	10000	Yes	0.16 (0.09)	0.18 (0.09)	19.01 (0.28)
1000	100000	Yes	5.79 (5.15)	5.95 (5.19)	304 (6.74)
10000	1000000	Yes	724 (717)	742 (726)	-
100000	1000000	Yes	1046 (1030)	1060 (1041)	-
1000000	1000000	Yes	27.71 (2.61)	30.30 (2.59)	out of memory

Conclusion

- Simulation plays an important role in distributed systems research, development and education
- SimCore framework is aimed to provide a solid foundation for building simulation models of distributed systems and beyond
  - Domain-agnostic, generic event-driven programming model
  - Callbacks and async mode to conveniently model any execution logic
  - High performance and ability to simulate large-scale systems
- The framework applicability and versatility are demonstrated by building several common and domain-specific simulation libraries

Code and documentation: https://github.com/systems-group/simcore