Concurrent Programming with Actors and Microservices

Maximilian Irro

Seminar für DiplomandInnen
5.11.2018

Forms of Concurrent Execution

Forms of Concurrent Execution

- **Pseudo-Simultaneous**: in alternation on a single CPU

Forms of Concurrent Execution

- **Pseudo-Simultaneous**: in alternation on a single CPU
- **Parallel**: truely simultaneous on several CPU cores

Forms of Concurrent Execution

- **Pseudo-Simultaneous**: in alternation on a single CPU
- **Parallel**: truely simultaneous on several CPU cores
- **Distributed**: several host machines

Foundational Issues of Concurrent Programming

Foundational Issues of Concurrent Programming

- **Expression of concurrent execution**: threads, futures, coroutines, etc.

Foundational Issues of Concurrent Programming

- **Expression of concurrent execution**: threads, futures, coroutines, etc.
- **Communication**: shared state vs. message passing

Foundational Issues of Concurrent Programming

- **Expression of concurrent execution**: threads, futures, coroutines, etc.
- **Communication**: shared state vs. message passing
- **Synchronization**: semaphores, locks, STM

Programming Abstractions

Programming Abstractions

- **Language-Construct Approach**: threads + locks

Programming Abstractions

- **Language-Construct Approach**: threads + locks
- **Operating System Approach**: processes + pipes

Programming Abstractions

- **Language-Construct Approach**: threads + locks
- **Operating System Approach**: processes + pipes
- **Network Approach**: processes + network channel

Actor Model

Actor Model

- Defines theoretically well-known constructs
- Receive and process messages (asynchronous, passiv)
- Process one message at a time
- Encapsulate state exclusively
- Runtime system executes actors concurrently
- Single-threaded semantics internally: exclusive state ownership + isolated message processing

Microservices Paradigm

Microservices Paradigm

- Complex functionality through composition of several *services*
- Microservice: small, independent executable
- „small“ size $\rightarrow$ in it's *scope of responsibility*
- Every microservice a dedicated operating system process
- Executed by an operating system scheduler (concurrency/parallelism)
- Communicate via message passing channels
- Network-based communication $\rightarrow$ distribution

Research Questions

Research Questions

RQ1 Why do actors and microservices qualify for programming concurrency?

RQ2 How do the actor and the microservice model facilitate concurrent execution?

RQ3 What are the expressive capabilities of actors and microservices regarding concurrent programming concerns?

RQ4 How does the performance of actors and microservices compare in a multi-core environment relative to a concurrent system scenario?

RQ1: Why do actors and microservices qualify for programming concurrency?

RQ1: Why do actors and microservices qualify for programming concurrency?

- Encapsulate state exclusively $\rightarrow$ synchronization-free (**?**)
- No shared state $\rightarrow$ message passing communication
- Temporal + spacial decoupling $\rightarrow$ concurrent scheduling by runtime/OS

RQ2: How do the actor and the microservice model facilitate concurrent execution?

RQ2: How do the actor and the microservice model facilitate concurrent execution?

Actors

- Concurrent execution by actor runtime
- History of combining actors with other *compatible* concurrency abstractions (futures)

Microservices

- Concurrent execution by operating system
- Free to use *every* concurrency approach available to the technology stack internally

RQ3: What are the expressive capabilities of actors and microservices regarding concurrent programming concerns?

- For specific technology stack
- Actor variant: Akka
- Microservice: Spring Boot + Spring Cloud

RQ3: What are the expressive capabilities of actors and microservices regarding concurrent programming concerns?

	Actors	Microservices
Encapsulation	libraries face issues	process memory boundaries
Communication	async. primitive and abstractions on top	variety of channel technologies
Concurrent Exec.	by actor runtime + additional models	by operating system + every model available
Scalability	vertical scalability, horizontal scalability	

RQ4: How does the performance of actors and microservices compare in a multi-core environment relative to a concurrent system scenario?

- Benchmark system: domain-specific search engine

Benchmark System Architecture

- Gateway (G)
- CatalogStore (C)
- Updater (U)
- Web Crawler (W)
- Parser (P)
- IndexStore (I)
- Searcher (S)

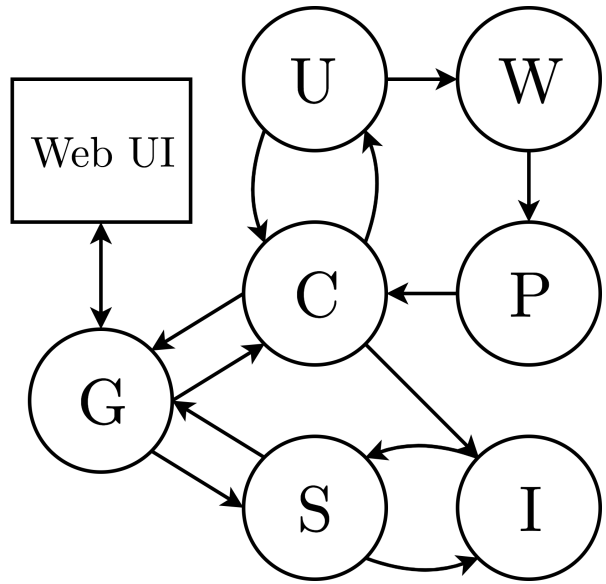


Figure 1: Interaction Model

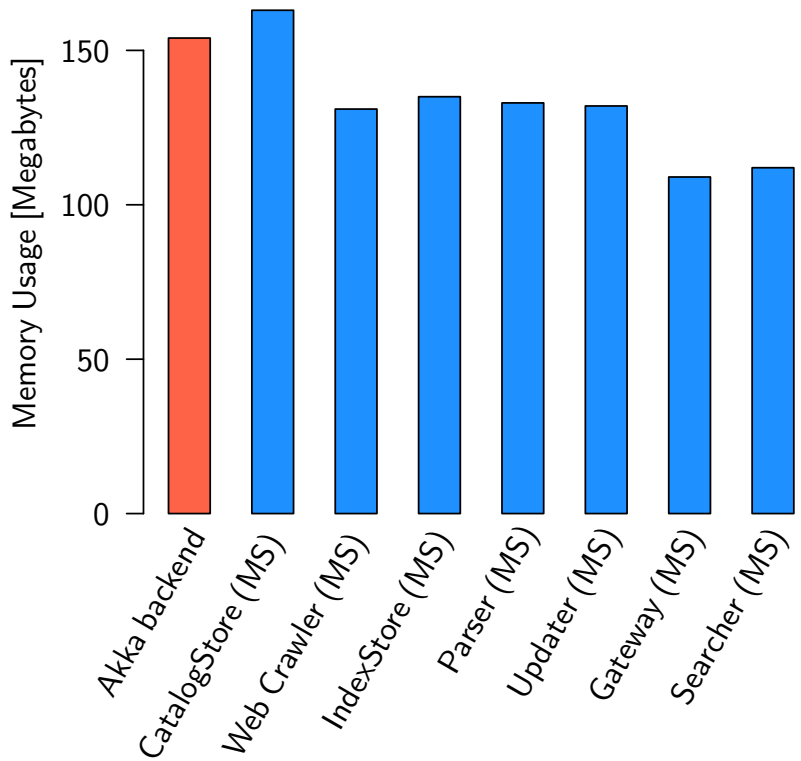
Software Artifact Analysis

Artifact	**LoC**	**sJAR (KB)**	**fJAR (KB)**	**Up (s)**
Akka monolith	4487	1004.3	76 775.1	5.5
CatalogStore (MS)	1838	56.1	89 225.8	14.6
IndexStore (MS)	724	23.8	83 518.2	8.8
Searcher (MS)	656	22.2	81 754.4	8.1
Web Crawler (MS)	716	23.5	83 517.9	9.2
Parser (MS)	703	24.2	83 519.1	8.6
Registry (MS)	334	9.9	90 699.7	9.4
Gateway (MS)	889	30.5	83 655.1	9.7
Updater (MS)	693	23.9	83 518.3	8.7

- $\sum \mathrm{LoC(MS)} = 6553$, about 46 % larger

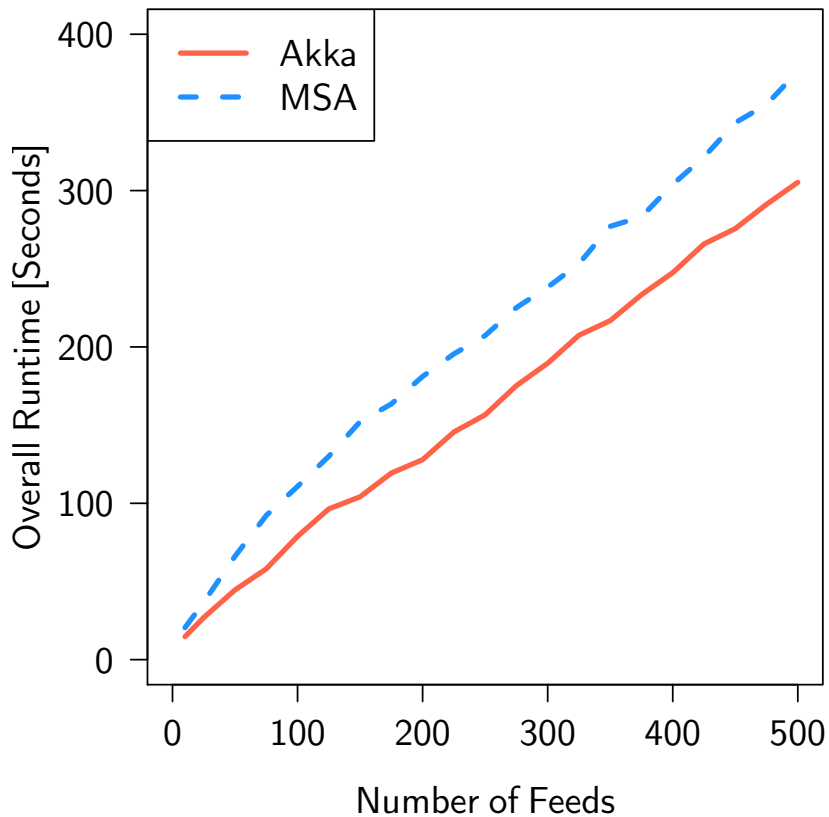
Artifact Memory Consumption

Memory consumption of the executable artifact VMs in the indexing phase:



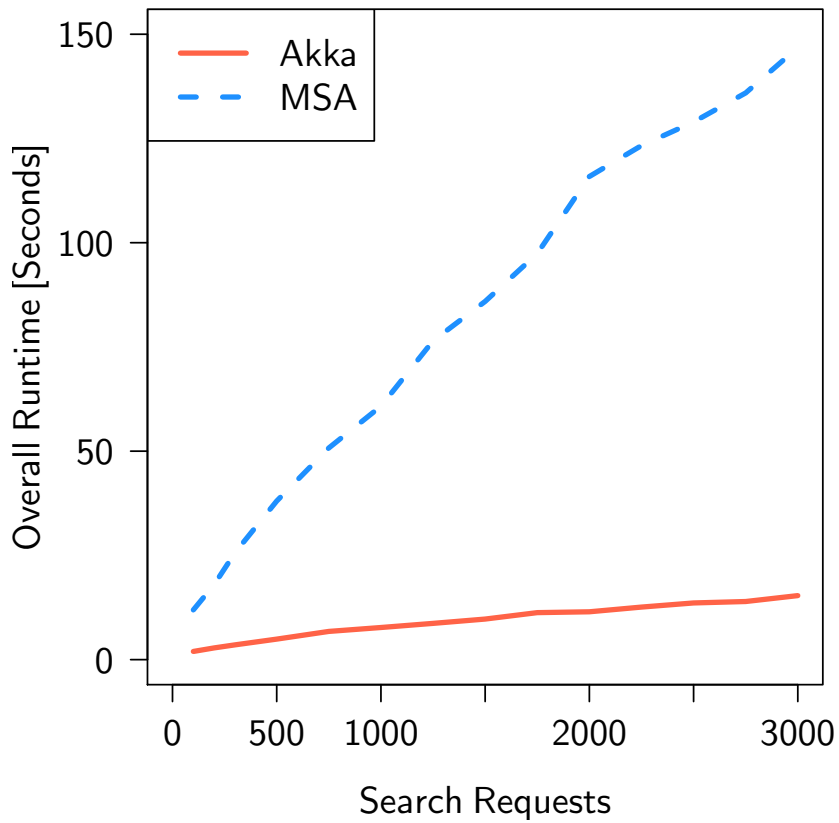
Overall Processing Time: Indexing Subsystem

Benchmark results for the overall processing time of the indexing subsystem:



Overall Processing Time: Retrieval Subsystem

Benchmark results of the overall processing time for the retrieval subsystem:



Contributions

- Compared the programming of concurrent computation with the actors and microservices
- Explored the interrelations of the two models and filled a gap in the literature
- Designed a non-trivial scenario for a concurrent domain-specific search engine
- Actor and microservice implementation
- Capability evaluation and efficiency benchmark

</end>

Supplemental: State Encapsulation vs. Isolation

- Microservice: process memory boundaries
- Actors (on the JVM):
  - Visibility + Accessibility $\rightarrow$ information hiding
  - Reference types + pass-by-value $\rightarrow$ immutability
  - Coding conventions required

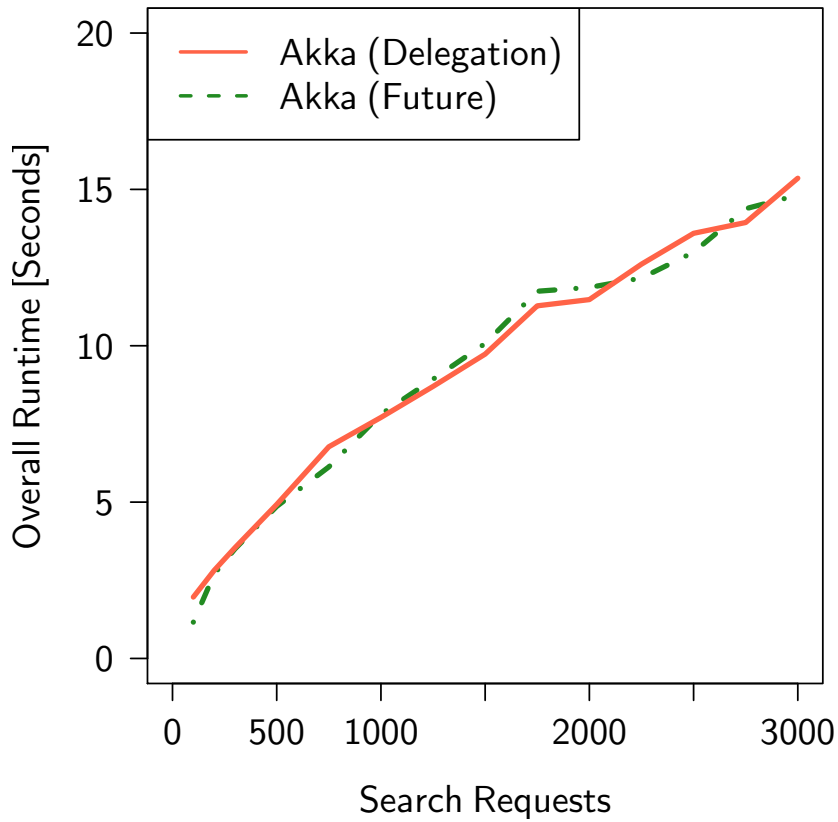
Supplemental: Actor state isolation in Java

```
public class Foo extends UntypedActor {
    public String bar;
    public static Props props() {
        return Props.create(Foo.class, () -> new Foo());
    }
    @Override
    public void onReceive(Object msg) {
      /* handle msg */
    }
}

final ActorRef foo = system.actorOf(Foo.props());
```

Supplemental: Semi-Synchronous Communication in Akka

Comparison of the benchmark results for the retrieval subsystem using either delegation or futures for request/response communication in the Akka:



Supplemental: Relevance of the Benchmark

- First benchmark comparing Akka actors and Spring- based microservices (full application context)
- Lack of different interaction modes in microservice architecture benchmarks according to literature $\rightarrow$ comparison to asynchronous actor system