Software AG
Freedom as a Service

USING ACTORS FOR MODULAR CONCURRENCY IN OSGI

BUILDING HIGHLY CONCURRENT SYSTEMS

Udo Hafermann
Todor Boev

ABSTRACT

- The Actor model is an architectural pattern designed to support high-scale concurrency without the need for locking constructs and with simple memory safety rules
- This talk discusses how to add support for the Actor concurrency model to the OSGi environment
- We want to retain the composition of OSGi services as the basic model for creating applications while at the same time allowing application developers to schedule concurrent execution with an actor runtime, rather than to use threads and locks
- We show a balanced way of combining both types of computation, each structured around a modularity construct with different properties: bundles for the blocking parts calling services, and actors for the non-blocking parts exchanging messages

OVERVIEW

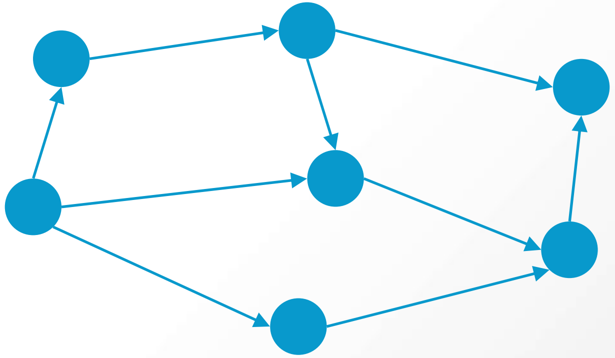
- Motivation
- The actor model
- Actors in Java
- Example: the dining philosophers
- Implementing the example
- Demo
- Findings and open ends

MOTIVATION

- More and more asynchronous concurrency in applications
- Primitives like PushStreams, Promises are functional and do not model state well
- Communication between stateful services with multiple threads and locks is hard to get right
- An alternative is asynchronous message passing
  - Which enables safe composition under concurrency
  - Of which actor systems are a pure embodiment
- So could actors be a complement to OSGi's modularity mechanisms to provide all-round concurrency capabilities?

THE ACTOR MODEL

- Actors process messages
- Message processing is defined by a behavior consisting of sequential steps
- In a step, an actor can send a message to an address
- In a step, an actor can create another actor and receive its address
- An actor can specify a new behavior to be used to process the next message

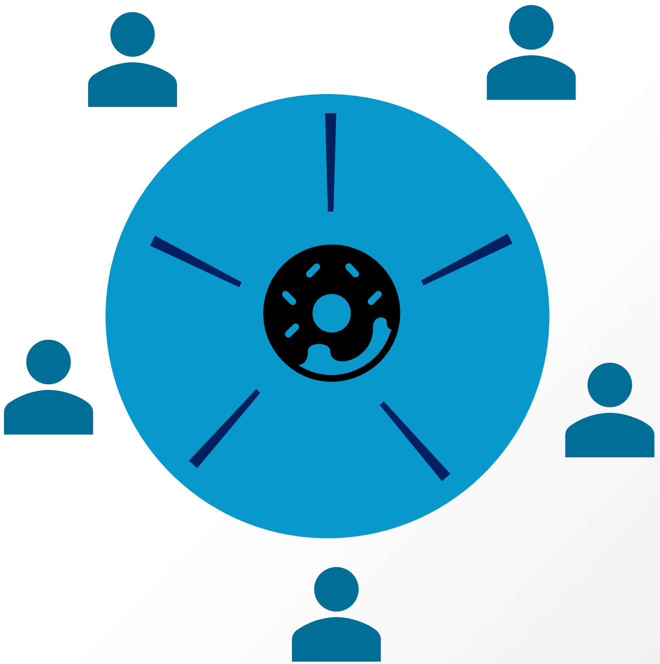


ACTOR IMPLEMENTATIONS (JAVA)

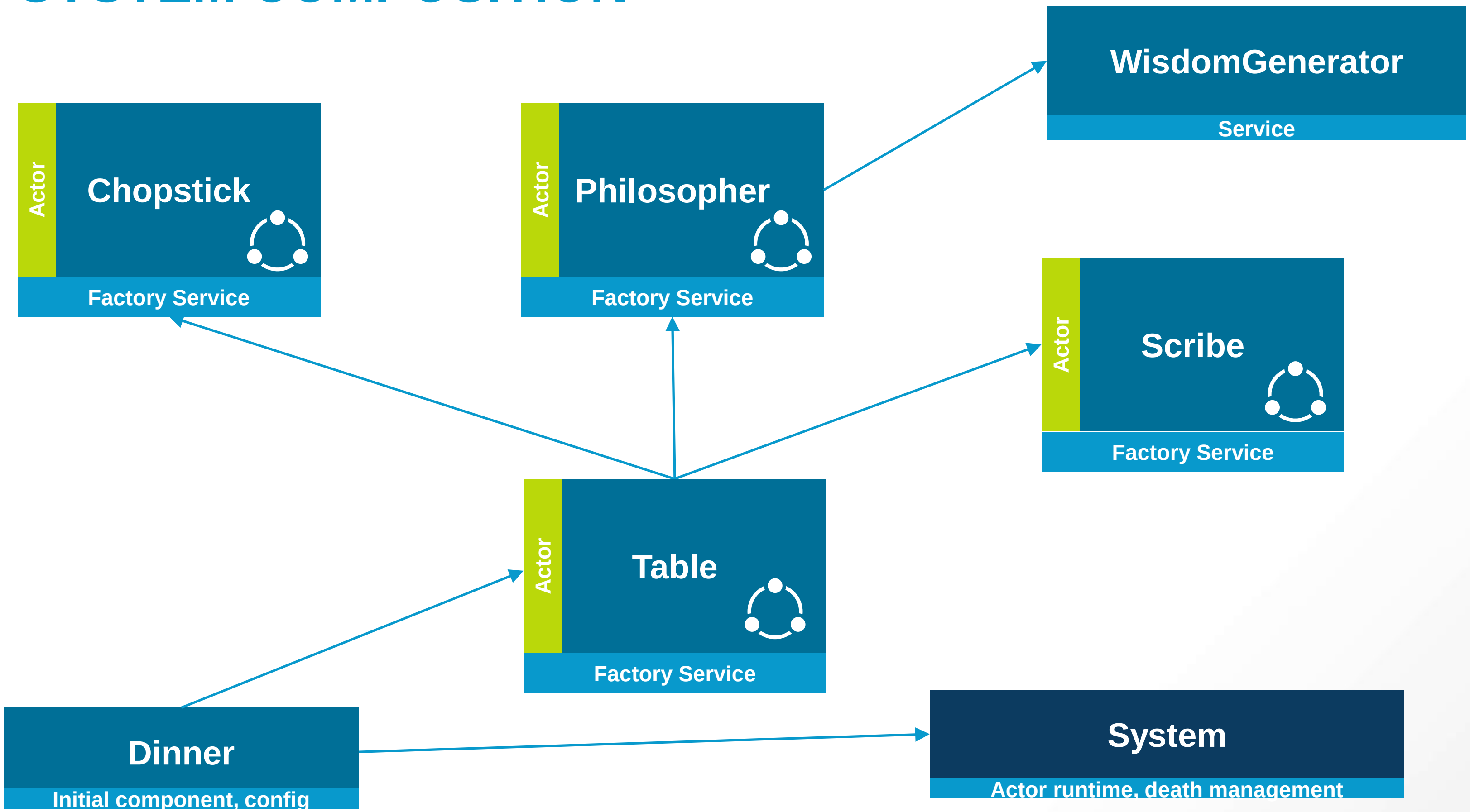
- Actor4J, Akka, µJavaActors, Jumi, Kilim, Kontraktor, Orbit, Quasar, …
- We chose Akka for our investigation
  - Feature-rich
  - Based on Scala, but with Java bindings
  - Actively developed and supported

THE DINING PHILOSOPHERS

- Each philosopher alternates between thinking and eating
- Each philosopher needs to wait for the two adjacent chopsticks to eat
- Before starting to eat, the philosopher shares new wisdom
- After eating, the philosopher puts down both chopsticks



SYSTEM COMPOSITION



OSGi Alliance

software AG
Freedom as a Service

COMPOSITION: DINNER

```
@Component(immediate = true)
public class Dinner {
        @Activate
        public Dinner(
                @Reference ActorSystem system,
                @Reference Table table,
                DinnerConfig config) {
```


OSGi Alliance

software AG Freedom as a Service

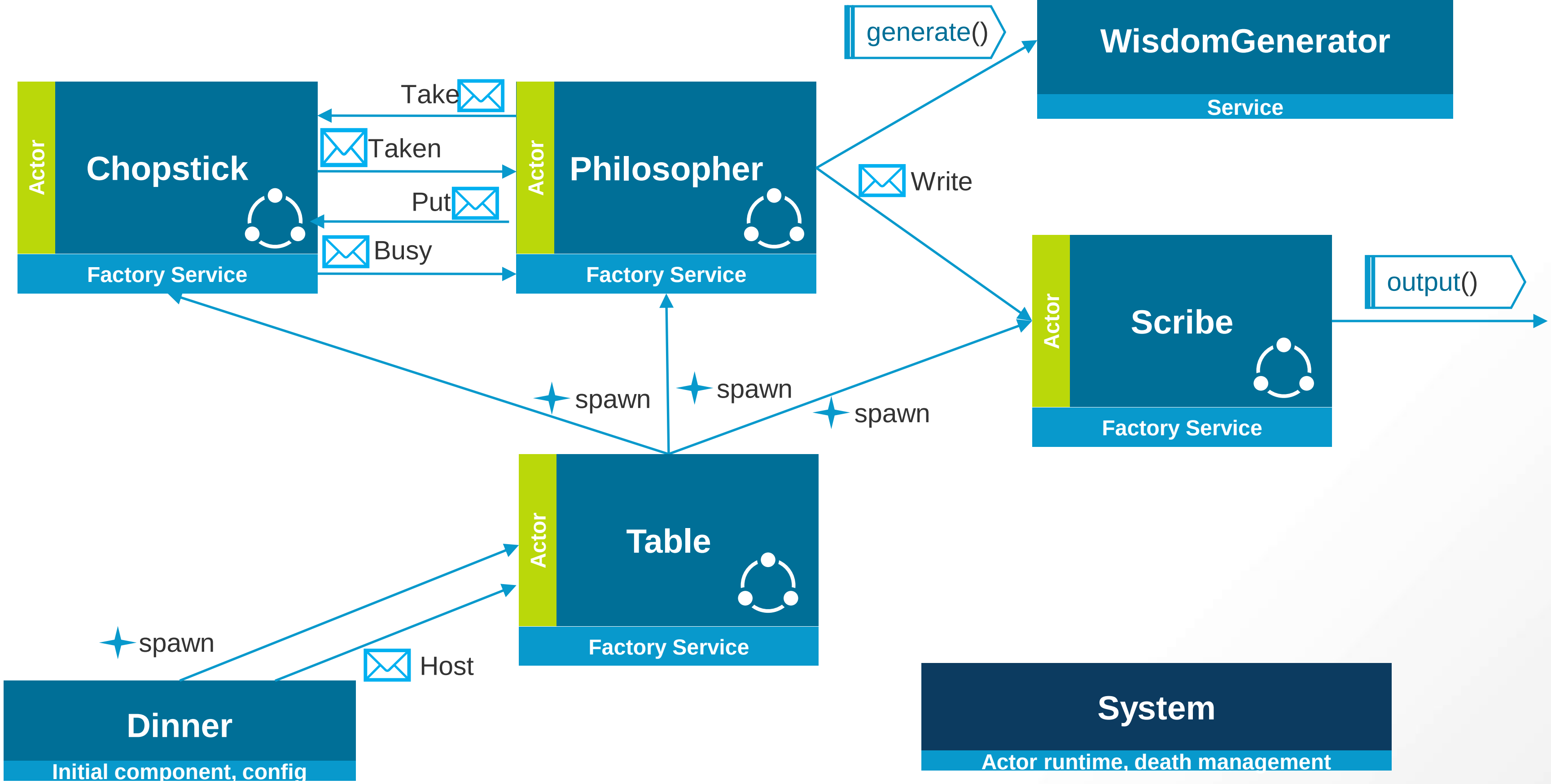
COMPOSITION: PHILOSOPHER

```
@Component
public class PhilosopherImpl implements Philosopher {
	@Activate
	public PhilosopherImpl(
		PhilosopherConfig config,
		@Reference WisdomGenerator wisdom) {
```


OSGi Alliance

software AG
Freedom as a Service

SYSTEM COMMUNICATION



COMMUNICATION: SPAWNING ACTORS

```
public Dinner(
                    @Reference ActorSystem<SpawnProtocol> system,
                    @Reference Table tables,
                    DinnerConfig config) throws Exception {
          table = spawn(system, tables.create(…),
                    tableName).toCompletableFuture().get();
…
```


COMMUNICATION: SPAWNING ACTORS

```
@Component
public class TableImpl implements Table {
        @Activate
        public TableImpl(
                @Reference Chopstick chop,
                @Reference Philosopher phil,
                @Reference Scribe scribe)
…
        onMessage(Host.class, (ctx, msg) -> {
                range(0, diners.size()).mapToObj(i -> {
                        String name = "chopstick-" + i;
                        return ctx.spawn(chop.create(), name);
```


COMMUNICATION: NON-ACTOR SENDS MESSAGE

```
@Activate
public Dinner(…) {
        table = spawn(system, tables.create(diners),tableName).toCompletableFuture().get();
        table.tell(new Table.Host(howLong));
```

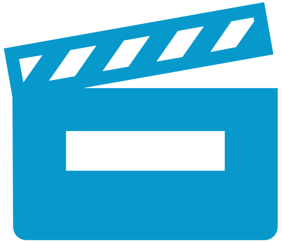

COMMUNICATION: MESSAGES BETWEEN ACTORS

```
@Component
public class ChopstickImpl implements Chopstick {
        private Behavior<Command> available() {
                .onMessage(Take.class, (ctx, msg) -> {
                        msg.replyTo.tell(new Taken(ctx.getSelf()));
                        return taken(msg.from);
        …
        private Behavior<Command> taken(ActorRef<?> owner) {
                .onMessage(Take.class, msg -> !msg.from.equals(owner), (ctx, msg) -> {
                        msg.replyTo.tell(new Busy(ctx.getSelf()));
                        return taken(owner);
        …
```


COMMUNICATION: ACTOR CALLS SERVICE

```
@Component
public class PhilosopherImpl implements Philosopher {
        private Behavior<Command> waitingForOtherChopstick(
                .onMessage(HandleTaken.class, (ctx, msg) -> {
                        CompletableFuture
                                .supplyAsync(wisdom::generate)
                                .thenAccept(res -> scribe.tell(new Scribe.Write(self, res)));
```


DEMO



FINDINGS

- It is possible to compose a system from actors and OSGi services
  - Injection works well
- Modular encapsulation of actors
  - Separation of APIs and implementation also for actors
- All communication to and from the non-actor world is asynchronous
  - Messages, promises, anyone can send messages
- Care is required to handle OSGi dynamics
  - Actor “death watch” synchronized with the service lifecycle

OPEN ENDS

- More in RFP 195
  - Persistence (for resilience)
  - Actor supervision (for error handling)
  - Better fusion of actors and services
    - e.g. avoid some boilerplate code

FEEDBACK


eclipsecon Europe 2019
OSGi™ Community Event 2019
LUDWIGSBURG, GERMANY | OCTOBER 21 - 24, 2019
EVALUATE THE SESSIONS
Sign in and vote using the conference app or eclipsecon.org
-1 0 +1

Freedom as a Service

