Peter Jurkovic

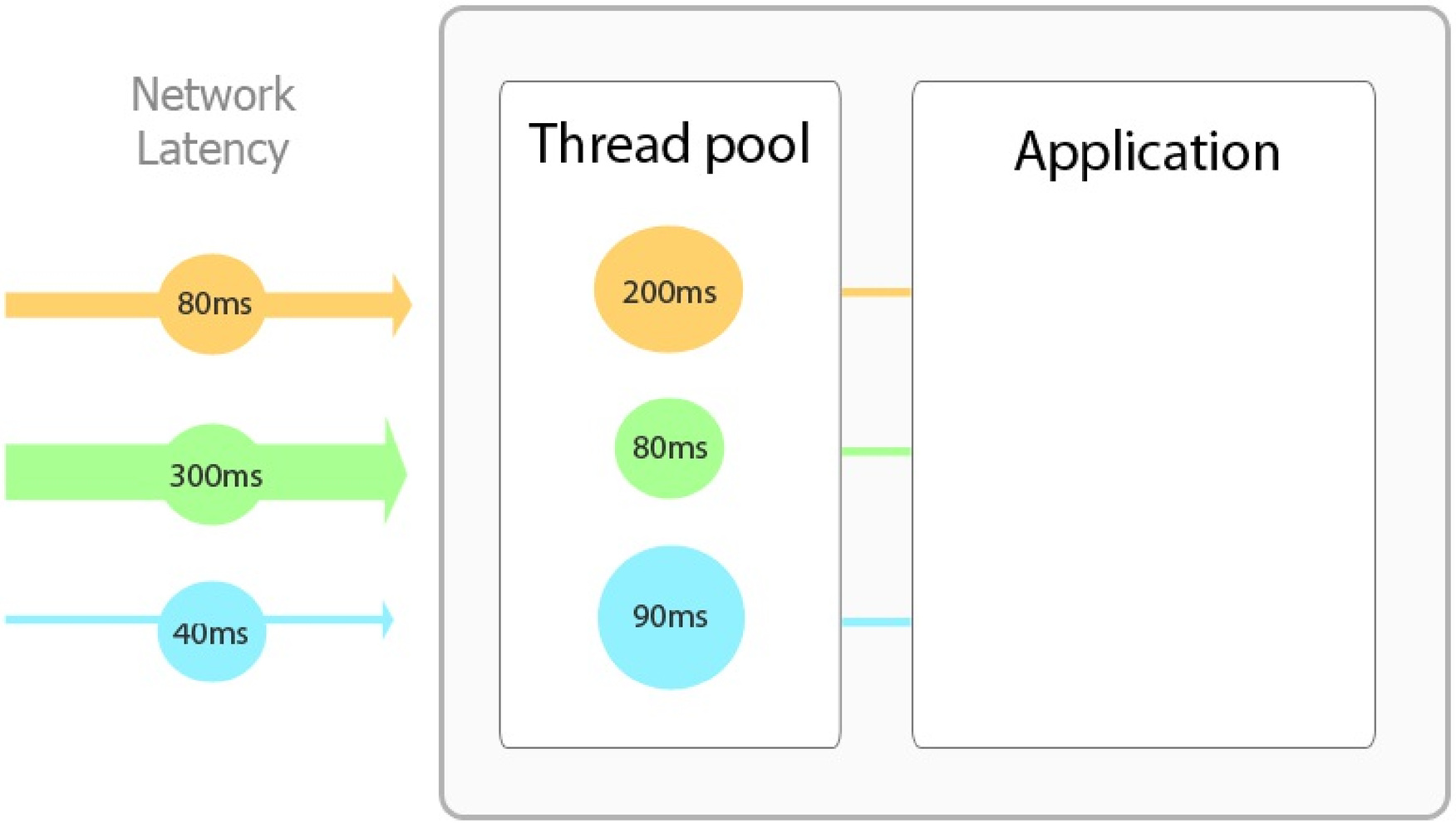
Agenda

- Motivation
- Ratpack
  - Request processing
  - Handlers
  - Registries & DI
  - Execution model
  - Application configuration
  - Application testing
- Reactive Streams

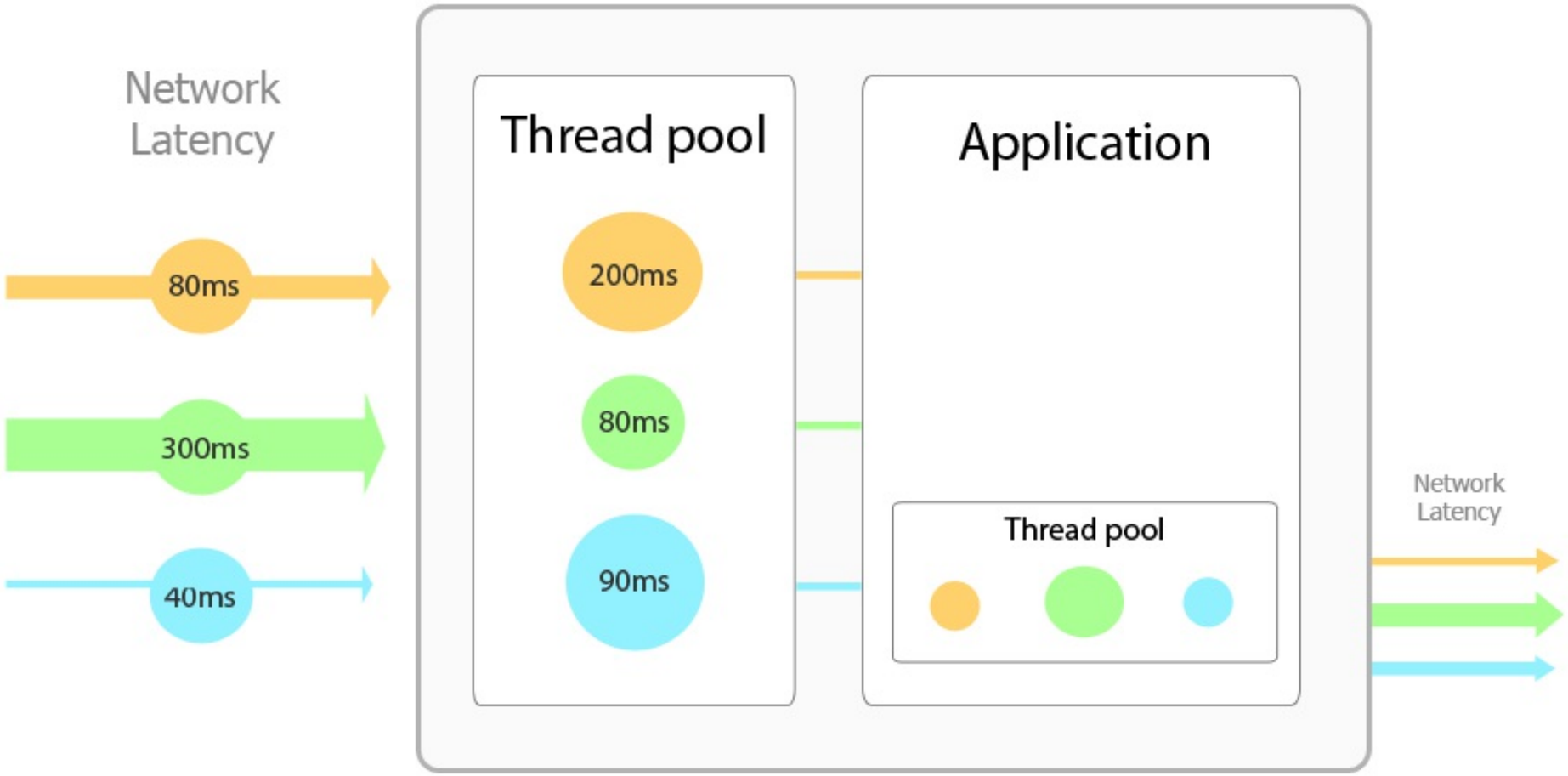
Java Servlets

- Born 20 years ago and still alive!
- Simple & effective processing model
- Request isolation
- But …

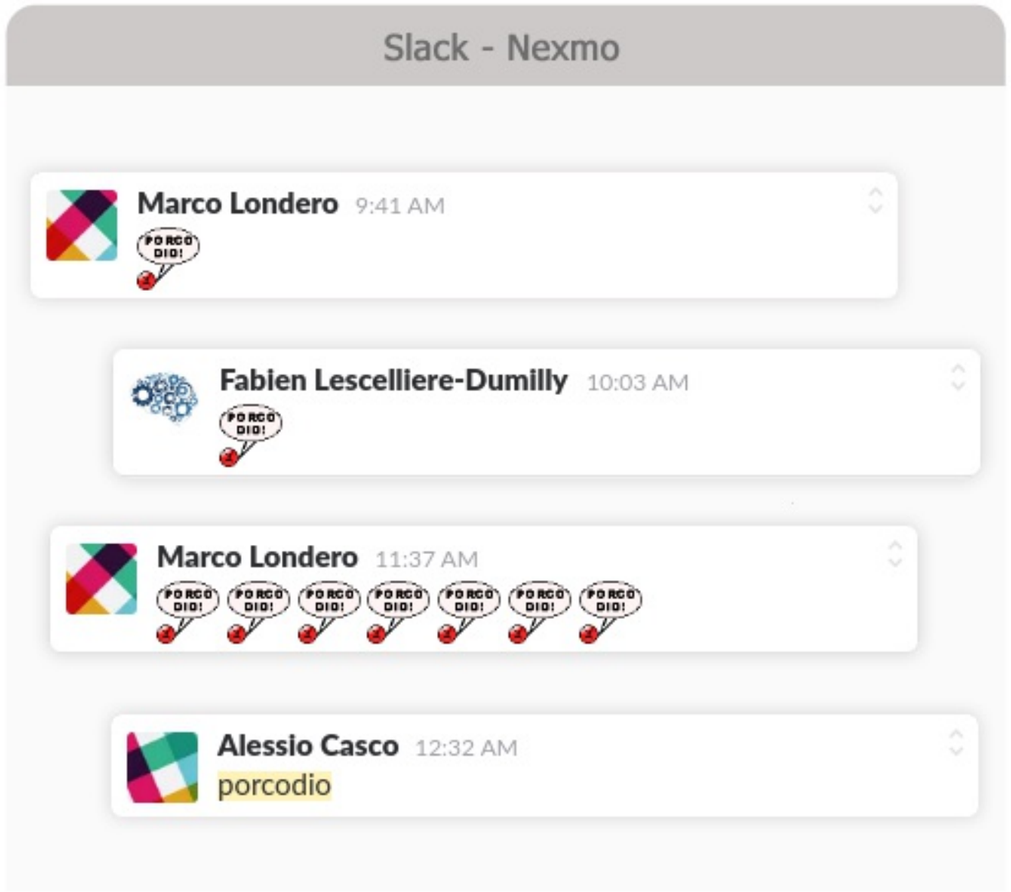
Servlet container



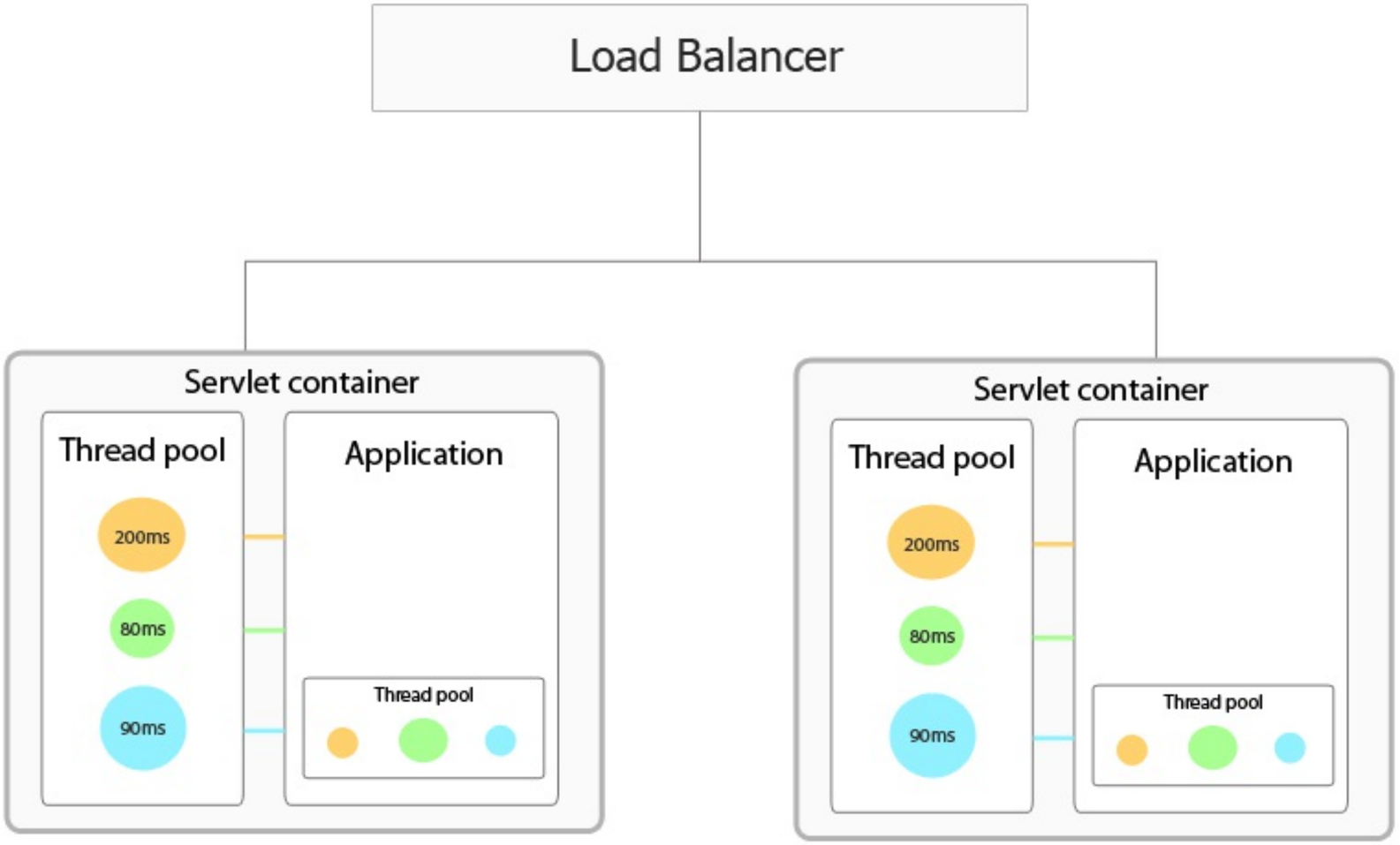
Servlet container



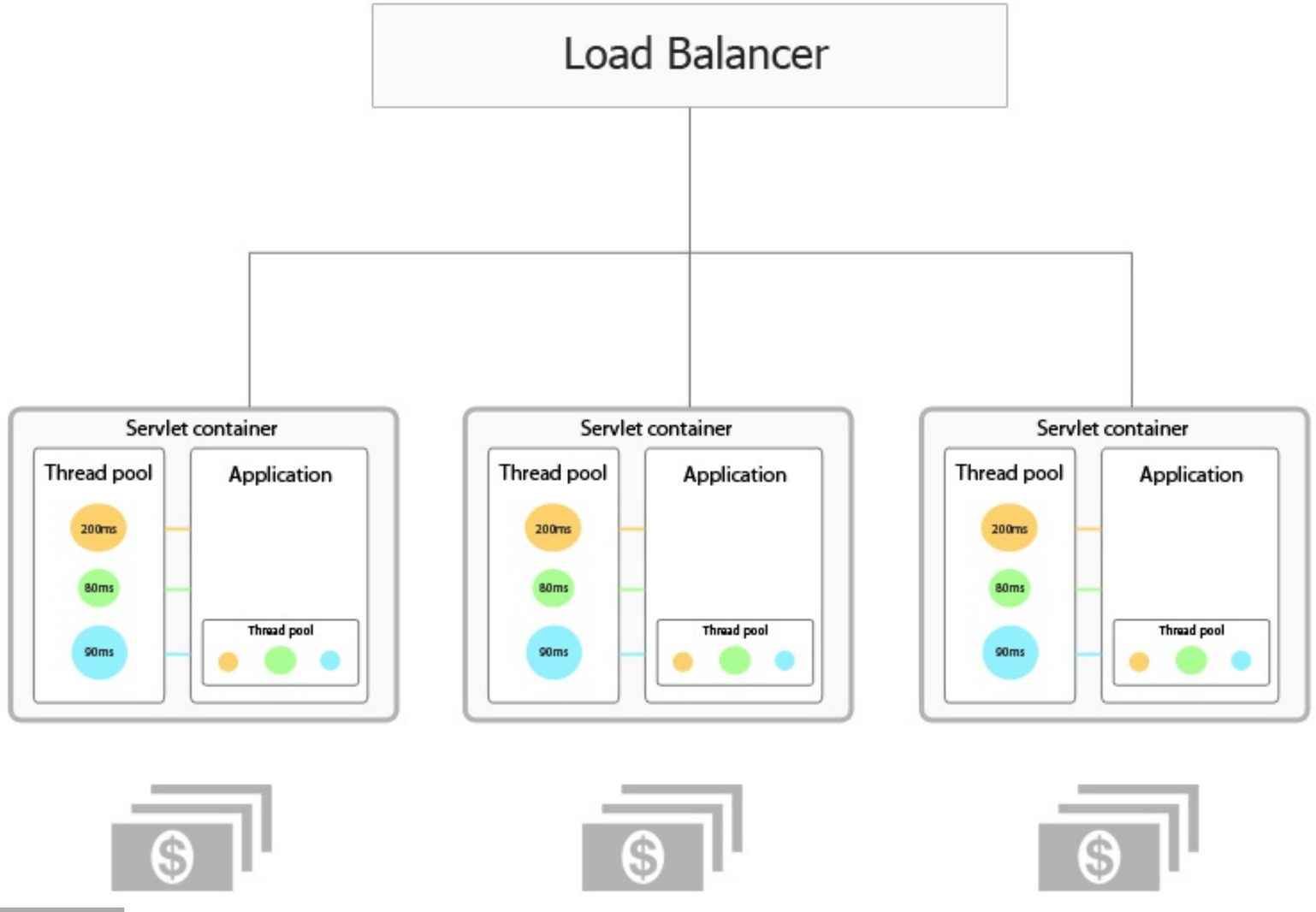
PORCODIO!


Slack - Nexmo
Marco Londero 9:41 AM
Fabien Lescelliere-Dumilly 10:03 AM
Marco Londero 11:37 AM
Alessio Casco 12:32 AM
porcodio

Scaling up

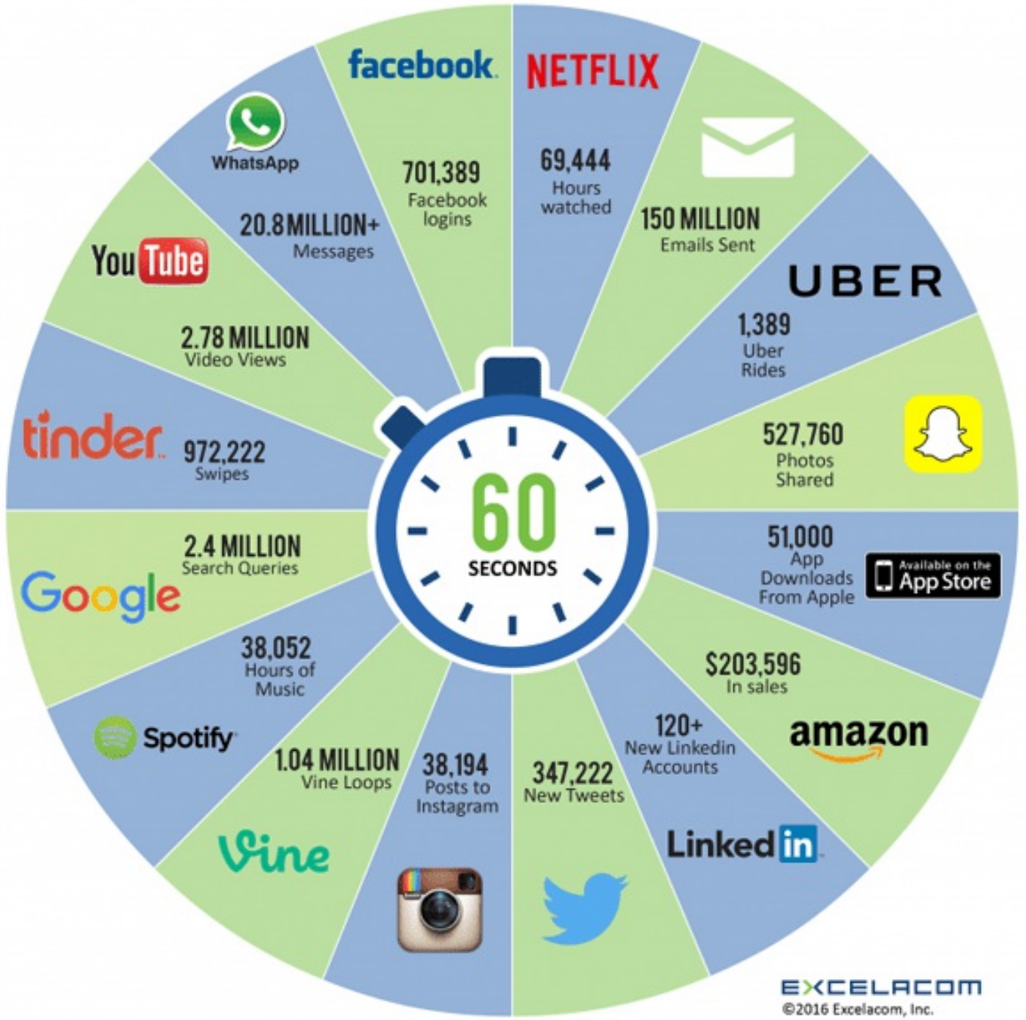

Load Balancer
Servlet container
Thread pool
200ms
80ms
90ms
Application
Thread pool
Servlet container
Thread pool
200ms
80ms
90ms
Application
Thread pool

Scaling up


Load Balancer
Servlet container
Thread pool
200ms
80ms
90ms
Application
Thread pool
Servlet container
Thread pool
200ms
80ms
90ms
Application
Thread pool
Servlet container
Thread pool
200ms
80ms
90ms
Application
Thread pool

Do we need change?

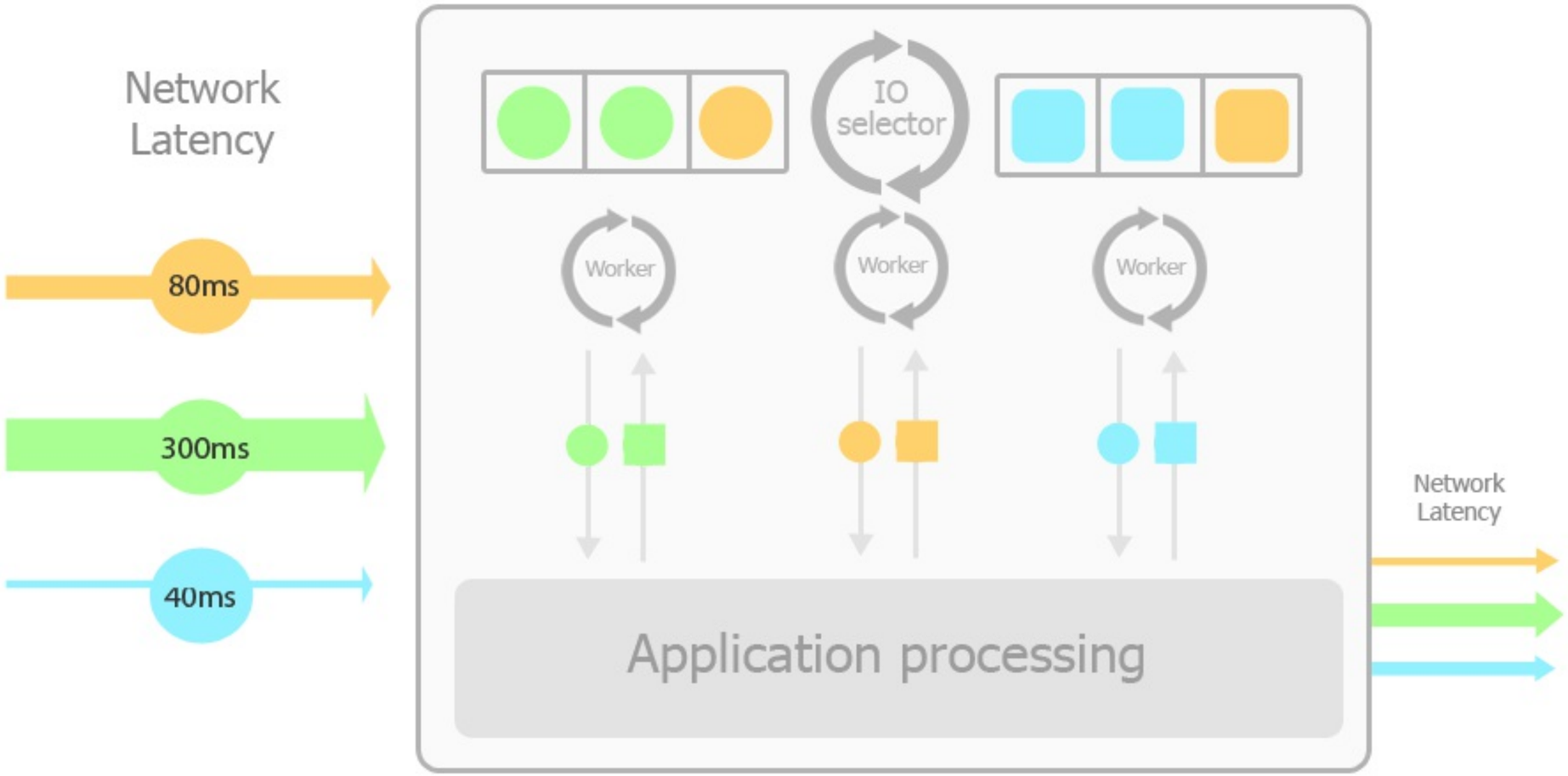
2016 What happens in an INTERNET MINUTE?



EXCELACOM

Is there another way?

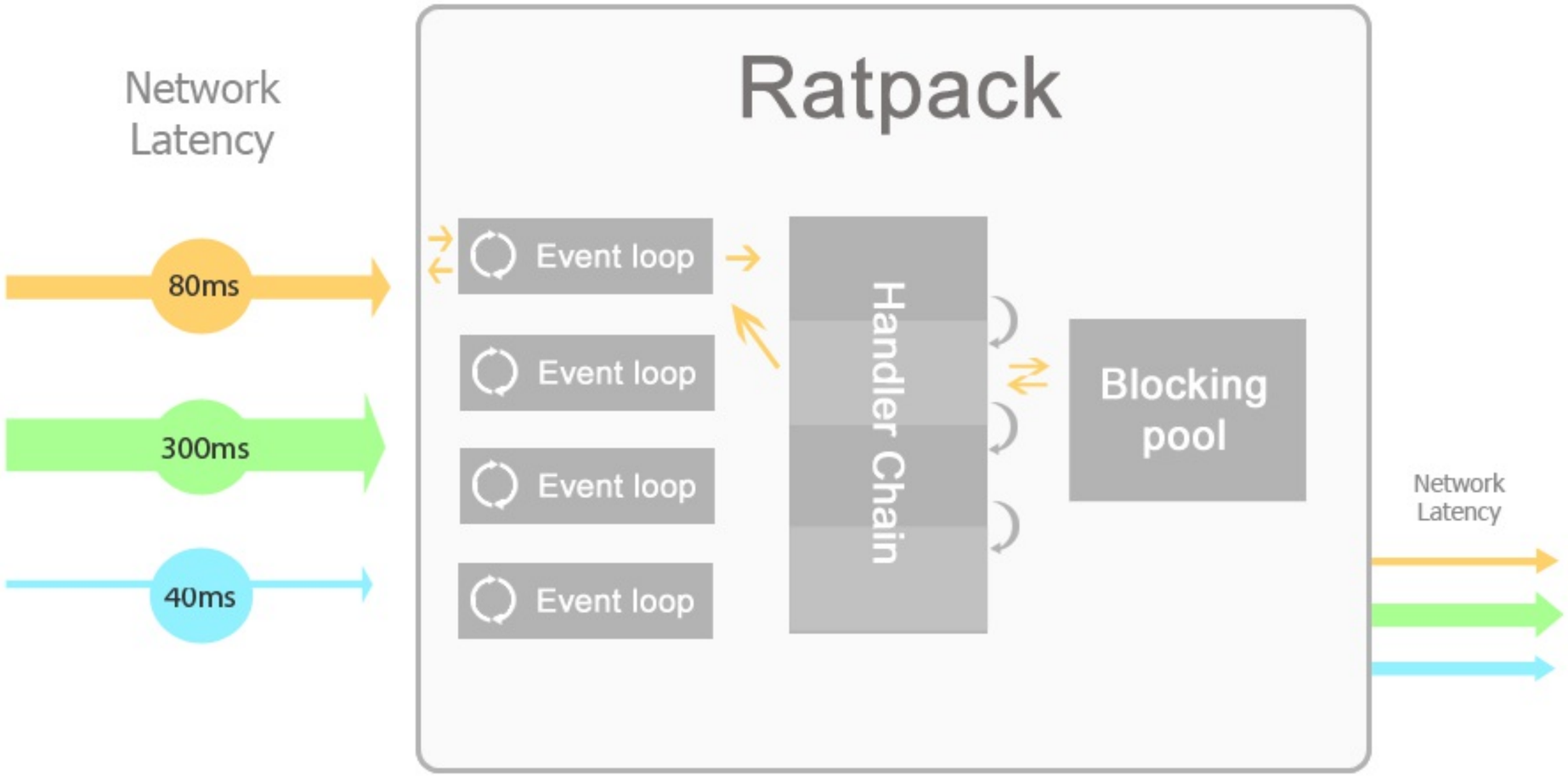
Non-blocking Runtimes



Ratpack

- High-performance, reactive web framework
- Build on Java 8 and Netty
- Using asynchronous non-blocking model
- Rapid develompent
- Aims to make async programiring on JVM easier

How does Ratpack work?



Ratpack - Hello Nexmo!

```
@Grab('io.ratpack:ratpack-groovy:1.4.2') 1

import static ratpack.groovy.Groovy.ratpack; 2

ratpack {
  handlers {
    get { 3
      render "Hello Nexmo!"
    }
  }
}
```

1. Groovy's dependency managment system "grab" the necessary dependencies and make it avaiable to our runtime classpath
2. Static import of the Groovy.ratpack method provides our script with the Domain-Specific Languages (DSL)
3. GET handler - binding a processing block for incoming HTTP GET requests

Handler Chain

- Handler is a "core" object in Ratpack
- Acts on a Context - can handle, delegate or add some new handlers
- Processed top-down (Order matters)

```
public class Server {

  public static void main(String[] args) throws Exception {
    RatpackServer.start( serverSpec -> serverSpec
        .handlers( chain -> chain
          .get( ctx -> {
            ctx.render("Hello Nexmo!");
          })
      )
    );
  }

}
```

Handler Chain (Continued)

- Can be defined to match on HTTP verb, URL or Content-Type
- Just a simple @FunctionalInterface
- Supports decoupling the rendering logic by implementing Renderer<T>

```
public class HelloNexmoHandler implements Handler{

  @Override
  public void handle(Context ctx) throws Exception {
    ctx.render("Hello Nexmo!");
  }

}
```

Handler method binding

- byMethod - for HTTP method
- byContent - for MIME types
- Easy & descriptive way how to build up REST api

```
handlers {
    prefix("api") {
        all { ... }
        delete { ... }
        post { ... }
        put { ... }
        patch { ... }
        get (":id") {
          byContent {
            html { ... }
            json { ... }
            xml { ... }
            noMatch { ... }
          }
        }
    }
}
```

Registries

- Chain is backed by registry
- "Map like type" where Key is a type and Value an instance
- Can be layered or "Joined"
- Something like ApplicationContext in Spring

```
handlers( chain -> chain
  .all( ctx -> {
    String userAgent = ctx.getRequest().getHeaders().get("User-Agent");
    ClientVersion clientVersion = ClientVersion.ofUserAgent(userAgent);
    ctx.next(Registry.single(clientVersion));
  })
  .get( ctx -> {
    ClientVersion clientVersion = ctx.get(ClientVersion.class);
    // ...
  })
)
```

Dependency injection

- Ratpack itself provides "sort of" DI
- Supports Google Guice
- Supports Spring

```
public class Server {
  public static void main(String[] args) throws Exception {
      RatpackServer.start( serverSpec -> serverSpec
        .registryOf( registrySpec -> registrySpec
          .add( UserService.class, new AsyncUserSerivce() )
        )
        .handlers( chain -> chain
          .get("user/:id", ctx -> {
            Long id = ctx.getPathTokens().asLong("id");
            ctx.get(UserService.class).getById( id )
               .map(Jackson:json)
               .then(ctx::render);
          })
        )
      );
    }
}
```

Dependency injection - Groovy

- Out-of-the-box support for Guice
- Slightly different approach

```
ratpack {
  bindings {
    bindInstance(UserService, new AsyncUserService())
  }
  handlers {
    get("user/:id") { UserService userService ->
      userService.getById( pathTokens.id ).then { user ->
        render( toJson( user )  )
      }
    }
  }
}
```

Async programming

- No language-level support for continuations
- Non-deterministic data flow

```
public interface AsyncDatabaseService {
  void findByUsername(String username, Consumer<User> callback);
}

RatpackServer.start(spec -> spec
  .registry(...)
  .handlers(chain -> chain
    .get(":username", ctx -> {
      AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
      String username = ctx.getPathTokens("username");
      db.findByUsername(username, user -> {
        ctx.render( user );
      });
    })
  )
);
```

From imperative to async programming

```
public interface AsyncDatabaseService {
  void findByUsername(String username, Consumer<User> callback);
  void loadUserProfile(Long profileId, Consumer<UserProfile> callback);
}

RatpackServer.start(spec -> spec
  .registry(...)
  .handlers(chain -> chain
    .get(":username", ctx -> {
      AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
      String username = ctx.getPathTokens("username");

      User;
      db.findByUsername(username, u1 -> user = u1);
      db.loadUserProfile(user.getProfileId(), userProfile -> { NPE :-(
        ctx.render( userProfile );
      });
    })
  )
);
```

A "solution" - Nested async call

- Nested code blocks - difficult to read and mantain. "Callback hell"

```
public interface AsyncDatabaseService {
  void findByUsername(String username, Consumer<User> callback);
  void loadUserProfile(Long profileId, Consumer<UserProfile> callback);
}

RatpackServer.start(spec -> spec
  .registry(...)
  .handlers(chain -> chain
    .get(":username", ctx -> {
      AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
      String username = ctx.getPathTokens("username");

      db.findByUsername(username, u1 -> {
        db.loadUserProfile(user.getProfileId(), userProfile -> {
          ctx.render( userProfile );
        });
      });

    })
  )
);
```

Promises - A better approach

```
public interface AsyncDatabaseService {
  Promise<User> findByUsername(String username);
  Promise<UserProfile> loadUserProfile(Long profileId);
}

RatpackServer.start(spec -> spec
  .registry(...)
  .handlers(chain -> chain
    .get(":username", ctx -> {
      AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
      String username = ctx.getPathTokens("username");

      db.findByUsername(username).flatMap( user -> {   1
        return db.loadUserProfile(user.getProfileId());   2
      }).then( profile -> {
        ctx.render( profile );
      });

    })
  )
);
```

1. Once the data is avaiable, it is streamed to the provided function
2. Transform the value of the stream by chaining the two promisses

Promises vs Callbacks

```
db.findByUsername(username, u1 -> {
  db.loadUserProfile(user.getProfileId(), profile -> {
    db.loadUserFriends(profile.getFriendsIds(), friends -> {
      db.loadPhotosFromFriends(friends.getPhotoIds(), photos -> {
        ctx.render( photos );
      });
    });
  });
});
```

```
db.findByUsername(username).flatMap( user -> {
  return db.loadUserProfile(user.getProfileId());
}).flatMap( profile -> {
  return db.loadUserFriends(profile.getFriendsIds());
}).flatMap( friends -> {
  return db.loadPhotosFromFriends(friends.getPhotoIds());
}).then( photos -> {
  ctx.render( photos );
});
```

Ratpack Promise

- Representation of a potential value
- Say things about the value without having it
- Represent a processing construct that is much more aligned with the concept of a continuation - denotes a frame of the continuation
- During the invocation of each frame, the continuation is suspended until its operation returns a value (fulfilling the promise)
- The serialization of the async calls gives - deterministic control flow

Ratpack Execution Model

- Is an implementation of a continuation and Promise<T> represents distinct frames of the continuation.
- Processing stream is created from the async call
- The processing calls that make up a stream are placed into a stack

```
RatpackServer.start( spec -> spec
 .handlers( chain -> chain
   .get( ctx -> {
     print("1");
     Blocking.get( () -> {
       print("2");
       return blockingUserService.loadUsers();
     }).then( userList -> {
       print("3");
       ctx.render( userList );
     });
     print("4");
   })
 );
 // prints 1423
```

Ratpack Execution Model (continued)

- Ratpack's execution model guarantees that the exception being thrown is the logical outcome. In fact, the background operation is never initiated.
- Requests are smartly handled based on data, not on response-writing timeouts
- Ratpack is said to provide Thread affinity

```
RatpackServer.start( spec -> spec
 .handlers( chain -> chain
  .all( ctx -> {
    ctx.next();
    throw new Exception(":-)");
  })
  .get( ctx -> {
    Blocking.get( () -> {
      return blockingUserService.loadUsers();
    }).then( userList -> {
      ctx.render( userList );
    });
  })
);
```

Application configuration

- Many build in config sources (Properties, YAML, System properties, Enviroment variables)
- Parsed into a tree of Jackson's ObjectNodes
- Avaiable from registries (Type safe and injectable)
- Custom Configurationi Source (Database, Network)

Application testing

- All needed in ratpack-test module
- First-class support for Spock (BDD)
- Handler unit testing with ratpack.test.handling.RequestFixture
- Functional and intergration testing with ApplicationUnderTest
- Mocking external services with EmbeddedApp
- Impossitions allow controls over Registries and configuration

Unit Testing Standalone Handlers

- **Handler unit testing with** ratpack.test.handling.RequestFixture

```
@Test
@Parameters({
  "Microservice v1.0 | V1",
  "Microservice v2.0 | V2",
  "Microservice v3.0 | V3"
})
public void test(String actualUserAgent, ClientVersion expectedVersion){
  Action fixture = request -> request.header("User-Agent", actualUserAgent);

  HandlingResult result = RequestFixture.handle(handler, fixture);

  ClientVersion actualVersion = result.getRegistry().get(ClientVersion.class);
  assertThat(actualVersion).isEqualTo(expectedVersion);
  assertThat(result.isCalledNext()).isTrue();
}
```

Functional & Integration Testing

- It is Easy - Ratpack Server is embeddable
- Integration & Functional testing with ratpack.test.ApplicationUnderTest
- Impositions allows control over configuration and registries

```
ApplicationUnderTest aut = new MainClassApplicationUnderTest(Server.class);

@Test
public void test() {
  TestHttpClient httpClient = aut.getHttpClient().requestSpec(spec -> spec
      .getHeaders().set("User-Agent", "Microservice v2.0")
    );
  String responseText = httpClient.get().getBody().getText();
  Assertions.assertThat(responseText).isEqualTo("Handling version: V2");
}
```

Sessions

- No session by default
- All needed in ratpack-session module
- Designed with support for distributed sessions in mind
- Build-in implementations: In-memory, Cookie-based, Redis bcked
- Easy to implement your own by implementing SessionStore
- Used Java serialization by default
- Customizeable by implementing SessionSerializer

```
dependencies {
  compile ratpack.dependency("session")
}
```

Modules - Hystrix

„Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable. ”

An application that depends on 30 services where each service has 99.99%

> 99.9930 = 99.7% uptime
> 0.3% of 1 billion requests = 3,000,000 failures
> 2+ hours downtime/month even if all dependencies have excellent uptime.

```
dependencies {
  compile ratpack.dependency("hystrix")
}
```

Modules - RxJava

„RxJava is arguably the most popular of the reactive programming libraries available on the JVM ”

```
.get(ctx -> {
  RxJavaUserService userService = ctx.get(RxJavaUserService.class);
  Observable<User> users = userService.getAll();
  RxRatpack.promise(users).then(users ->
    ctx.render(json(users))
  );
})
```

```
dependencies {
  compile ratpack.dependency("rx")
}
```

Modules - other

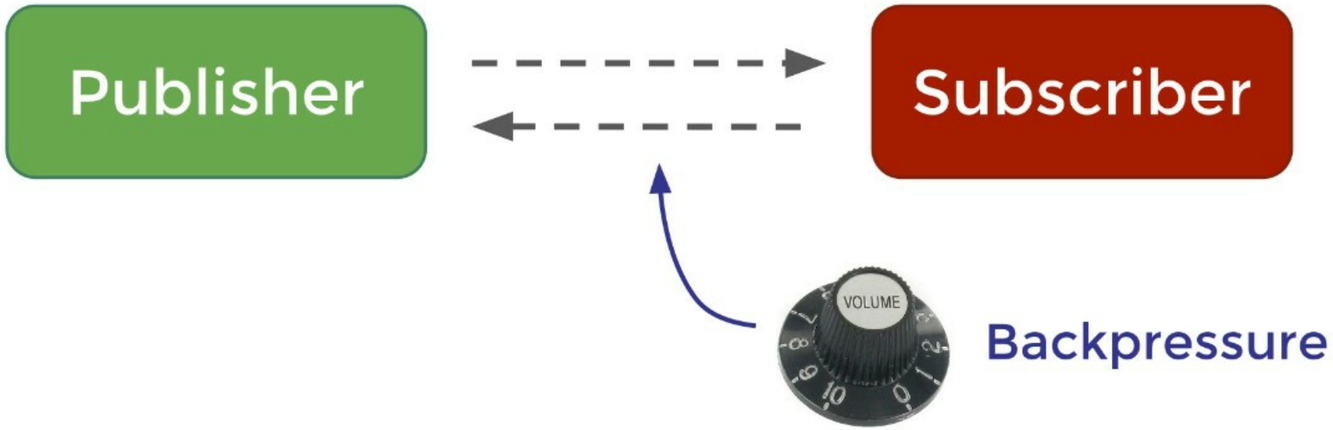
- Consul (Ask Esther)
- Hikari
- Retrofit2
- Dropwizard metrics
- Thymeleaf
- and more on https://github.com/ratpack

Demo time

Reactive Streams

„Reactive Streams is a protocol for asynchronous stream processing with non-blocking backpresure. ”

Stephane Maldini, Devoxx UK 2016



Sources & links

- BOOK: Learning Ratpack - O'Reilly Media (Dan Woods)
- Ratpack: Under the Hood (Luke Delay)
- Ratpack Web Framework (Dan Woods)
- There's so much more to Ratpack than non-blocking (Marcin Erdmann)
- Rapid Web App Development with Ratpack (Dan Hyun)
- Introduction to Reactive Programming (Stephane Maldini & Rossen Stoyanchev)
- Reactive programming (Stephane Maldini)
- Reactive Spring (Stephane Maldini & Rossen Stoyanchev)
- Project Reactor by Pivotal & Spring

Thank you