Peter Jurkovic

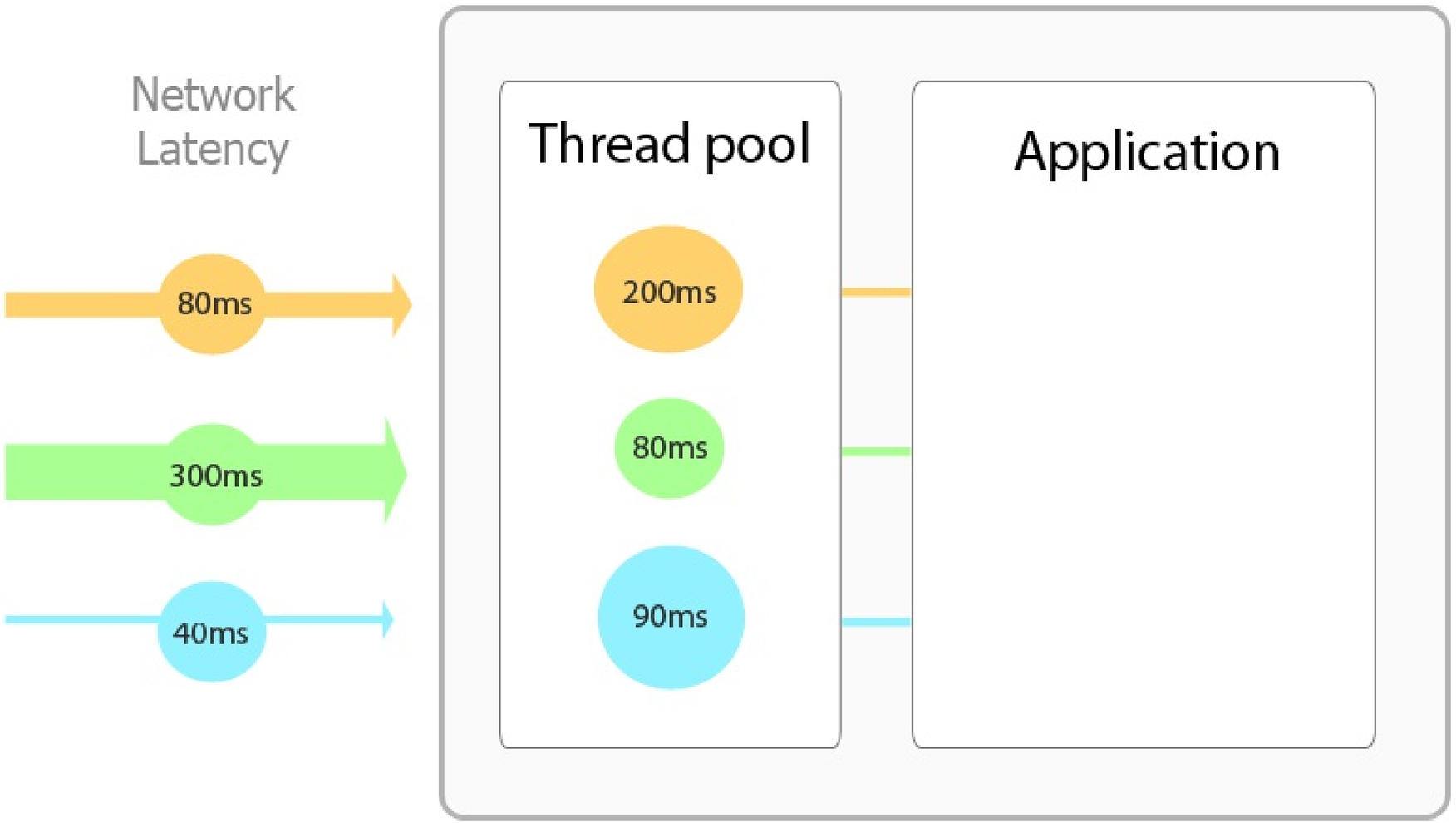
Agenda

- Motivation
- Ratpack - key concepts
  - Request handling
  - Dependency injection
  - Application configuration
  - Promises and Executions
  - Ratpack Modules
- Reactive Streams

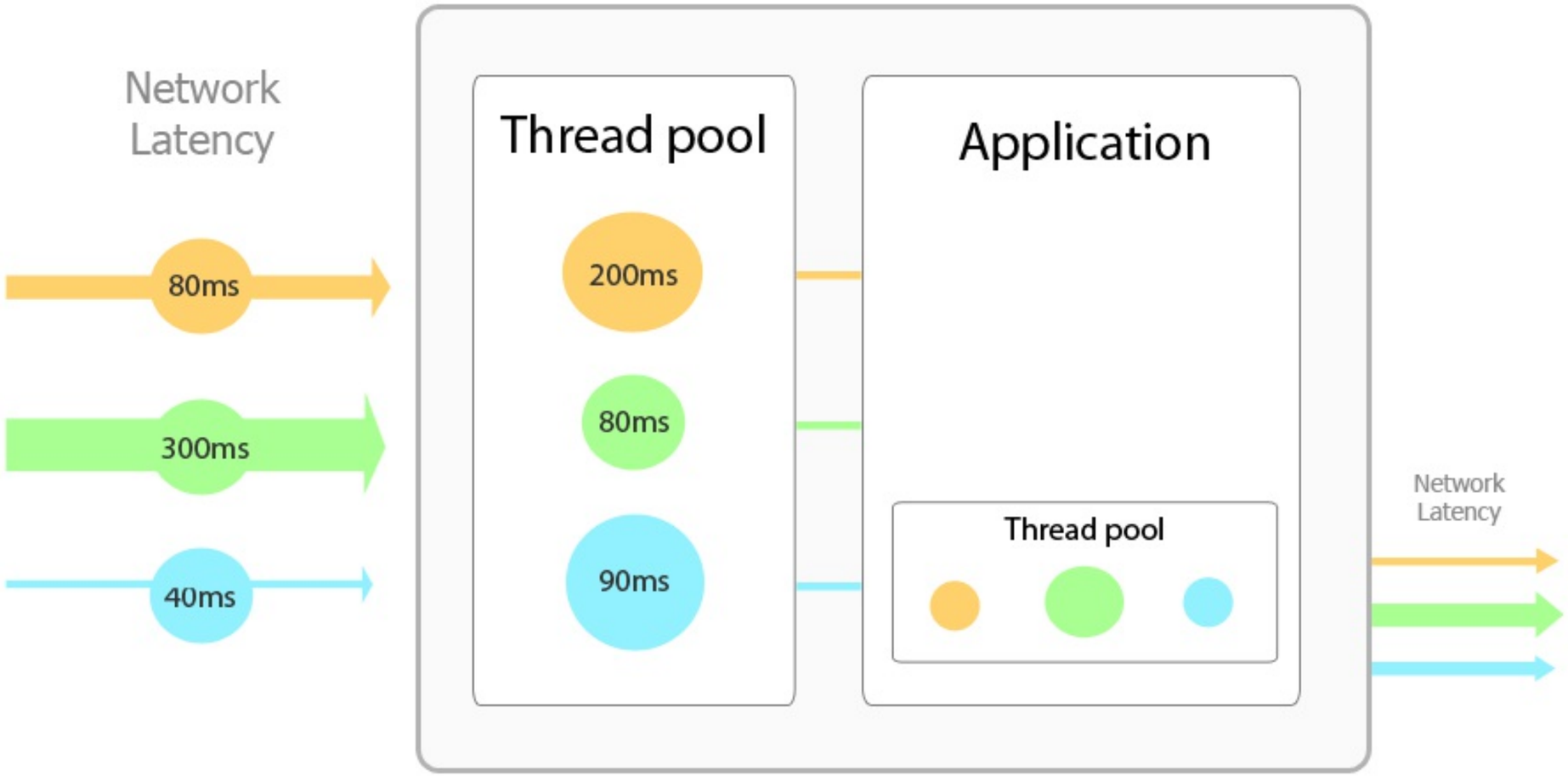
Java Servlets

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

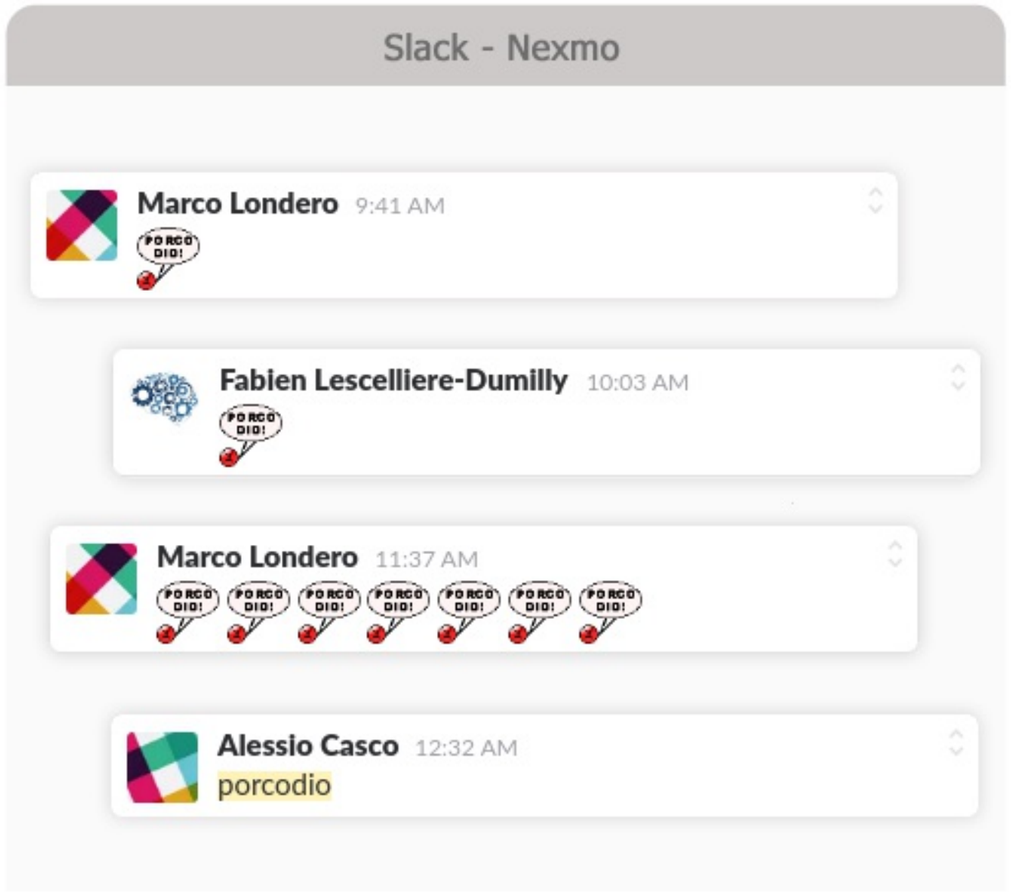
Servlet container



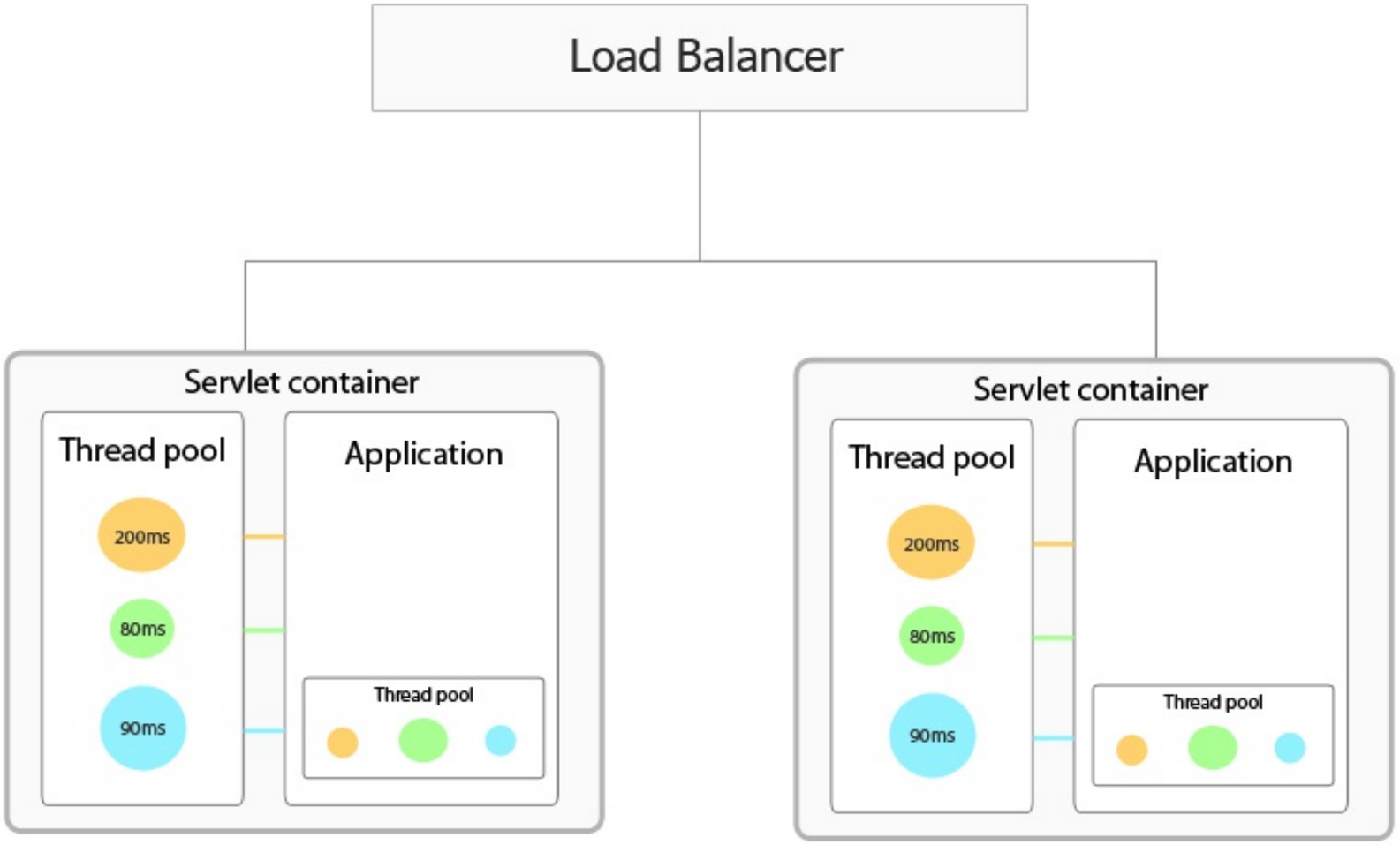
Servlet container



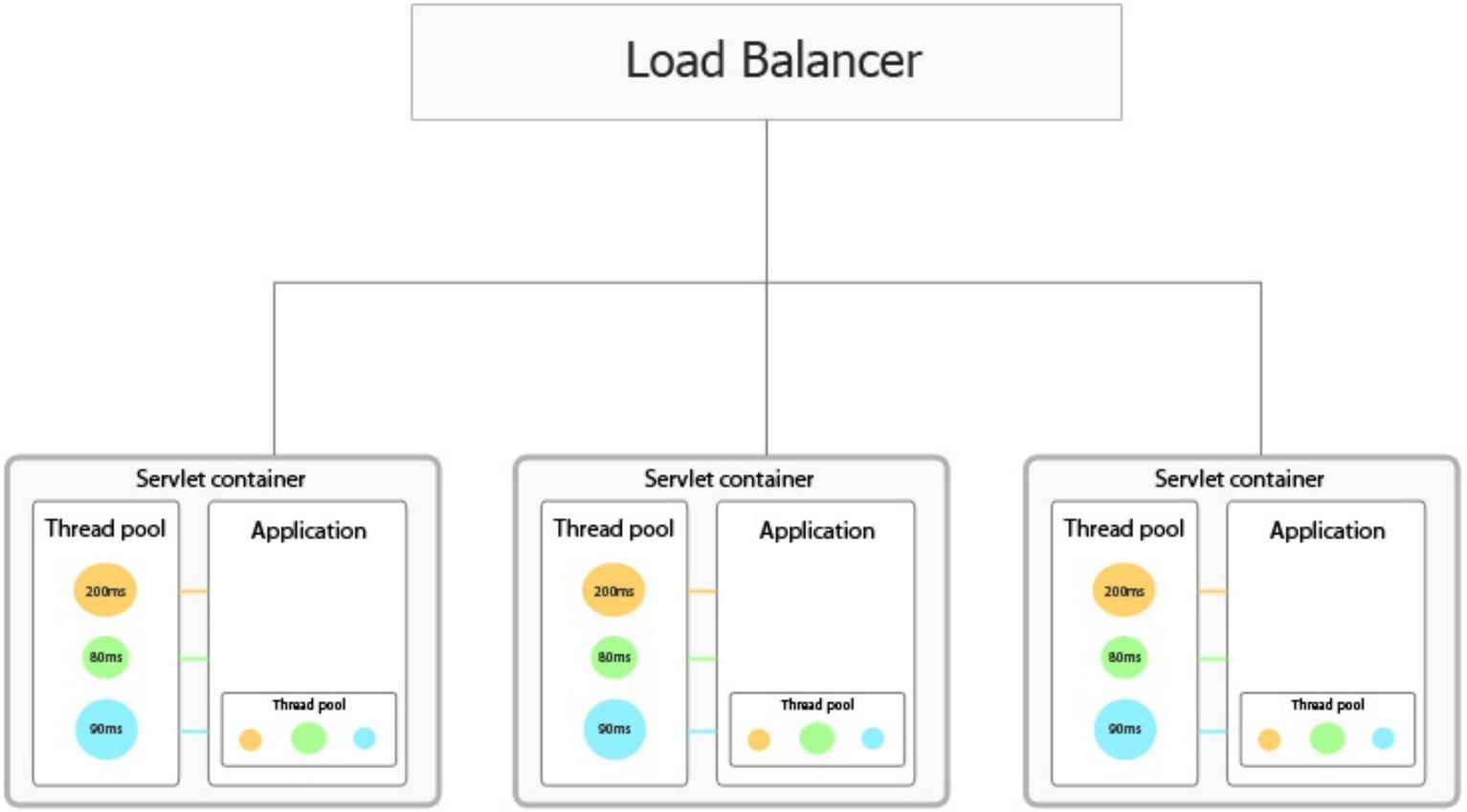
PORCODIO!


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

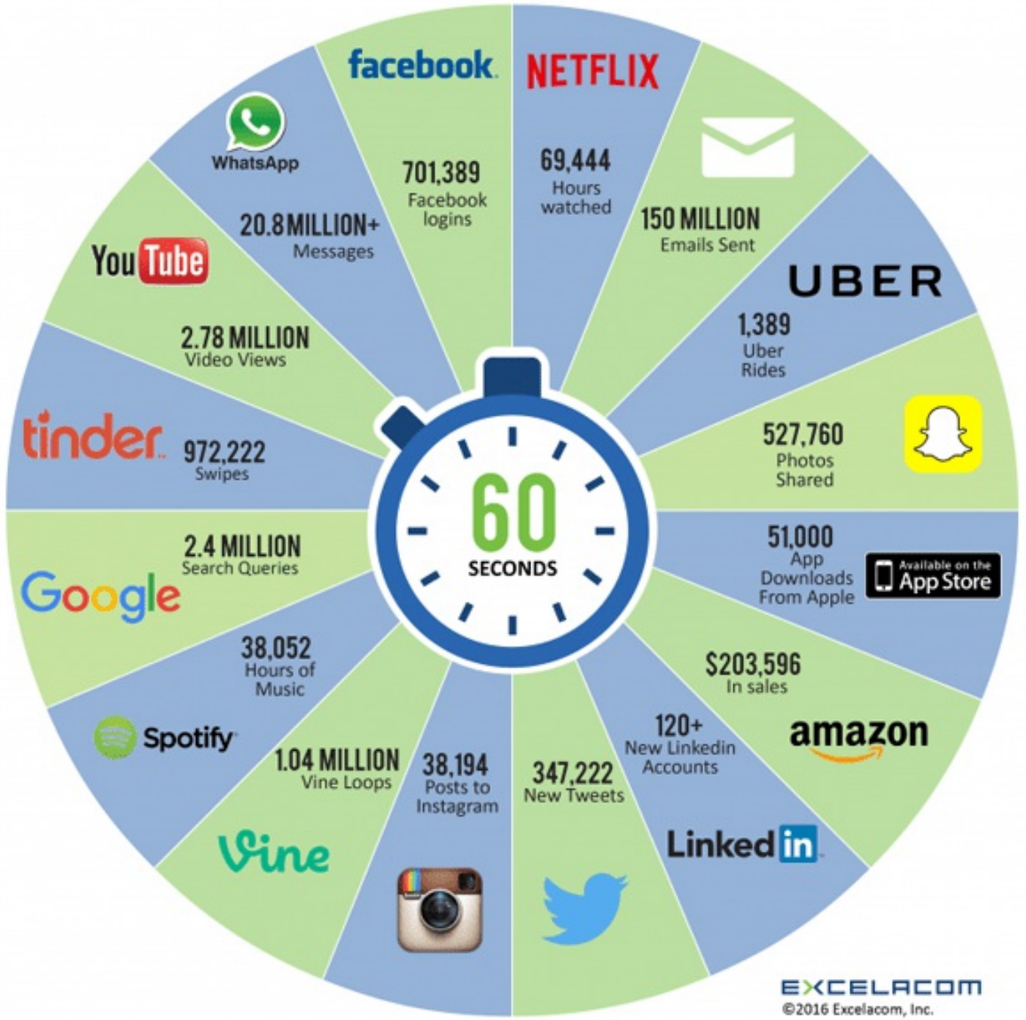

Load Balancer
Servlet container
Thread pool
200ms
80ms
90ms
Application
Thread pool
Servlet container
Thread pool
200ms
80ms
90ms
Application
Thread pool

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?

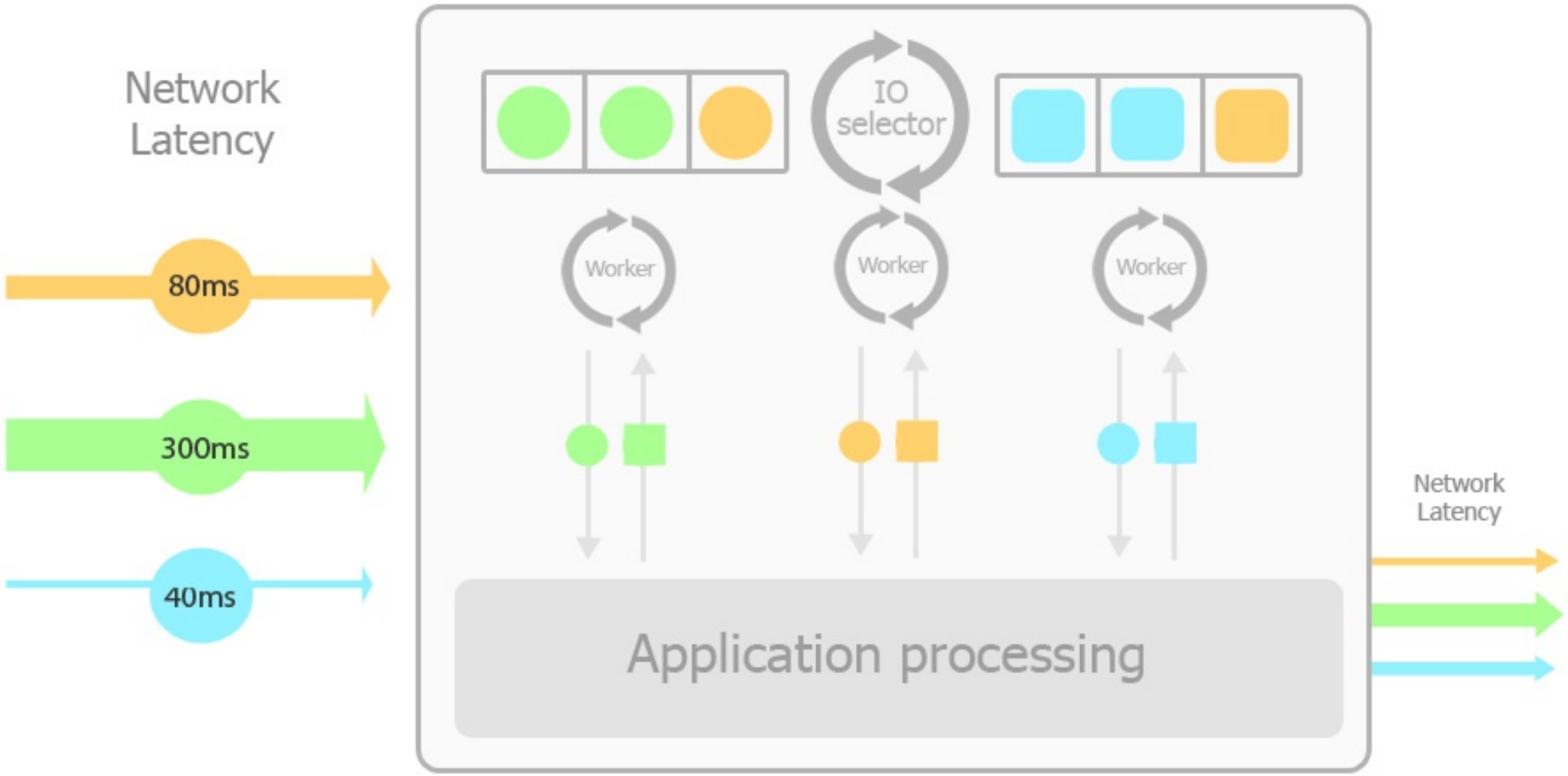
2016 What happens in an INTERNET MINUTE?



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

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 )
               .then( user -> ctx.render( json( user ) ) );
          })
        )
      );
    }
}
```

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 );
    });
  })
);
```