Master Thesis
Distributed Programming Laboratory

The Zone, a general model for asynchronous contexts

Valérian Pittet

ELCA supervisors

Philipp Oser

Benoît Briot

EPFL supervisor

Rachid Guerraoui

August 14, 2015

Abstract

This work aims to provide an asynchronous *context* to solve the most common problems and encountered needs when coding an asynchronous program. This *context* includes support for local values definition, asynchronous error catching, hooks into the asynchronous executions and primitives to build specialized features on asynchronous transitions.

The first step consists in analyzing the existing frameworks for asynchronous execution to determine the environments in which the code has to work. Then, a study of encountered challenges with asynchronous programming leads to a general solution, able to support the building of refined behaviors. Finally the implementation is integrated and tested on a real-world execution framework.

The developed solution addresses asynchrony concerns, but not concurrency. It takes inspiration from the zone.js and dart:Zone frameworks to build a similar concept for Java. It implements the required functionalities and specifies how to incorporate it into execution frameworks. Vert.x is used as case study to show concrete use cases and how the integration is realized.

Contents

Introduction

Who never faced the challenge of debugging an asynchronous program? And even the somewhat simpler task of following its execution and understanding its dependencies is neither easy, as illustrated in chapter 2.

The main reason of this difficulty is that our standard representations assume determinism and do not hold anymore with the non-determinism brought by the asynchrony. However, this non-determinism is precisely what we search when using asynchronous constructs: there is no way to bypass this problematic.

I propose a solution (chapter 4) inspired by the zone.js and dart:Zone[1] frameworks to implement functionalities that behave *uniformly* in both synchronous and asynchronous contexts. Moreover this solution aims to be applicable to any underlying asynchronous implementation, at least the most common ones presented in chapter 3. Chapter 6 shows how, unlike zone.js and dart:Zone, the Zone for Java implements primitive features, allowing to compose almost any other functionality. This way, the Zone steps down central features as error handling or asynchronous execution tracing to a standard Zone plug-in. It also allows a great flexibility to add aspects fulfilling needs discovered later.

Practical experiments (chapter 7) built on top of the Zone showed how it supports modularization of complex properties, for example tracing each asynchronous transition of the program. On the other hand, integrating the Zone in an execution framework is hard (consider many cases, avoid duplicate binding), but this work must be done once. Then, use the instrumented executor and Zones will work transparently on any program based on that framework.

The Zone is an innovative tool in that it addresses the problem of implementing uniform behaviors across context modifications, regardless of the synchrony. The Zone allows you to think in terms of transitions between asynchronous tasks and simply implement those transitions once, at a single place.

[1]Dart, a less popular web development language than JavaScript, is often referenced in this work, since it was one of the first to propose a Zone concept.

Motivations

To complete and give a clear idea of the problems I refer to, this chapter presents typical bugs encounteres in asynchronous programming.

2.1 Asynchronous Scope

First problem with asynchronous execution is dealing with contextual values. Consider this simplistic code:

```
Environment environment;

public void test() {
  environment = Environment.STANDARD;
  doStandardStuff();

  environment = Environment.CRITICAL;
  doCriticalStuff();

  environment = Environment.STANDARD;
  doStandardStuffAgain();
}
```

In synchronous execution, there is no problem and everything runs with the expected environment. Suppose now you need to improve performance. You change the method `doCriticalStuff()` code to run asynchronously and forget to care about `environment`. Very likely, `doCriticalStuff()` will run in a wrong environment since it may execute *after* `environment` recovers the `STANDARD` value.

This problem can be solved if we represent `environment` as a context persisting across asynchronous transitions.

2.2 Error Handling

If the problem of asynchronous scope seems too basic to require attention, error handling already brings more challenge.

```
public void test(int input) {

  try {
    heavyMathFunction(input);
  } catch (ArithmeticError e) {
    // handle error
  }

}
```

Again, as long as the code is synchronous there is no problem with the try-catch block. But once you decide to run asynchronously `heavyMathFunction(int)` to avoid application freeze, the code may not handle (or even see) a runtime error any more.

A solution to this problem needs ways to keep error handler across asynchronous transitions or report asynchronous error to the wrapping error handler.

2.3 Error Tracing

Suppose now asynchronous error are caught. Debugging unexpected errors stays hard. The stack trace only goes back to the beginning of asynchronous execution. If the cause of the error precedes the asynchronous code submission, nothing can track it back.

```
public void test() {

  int input1 = getInput1();
  startAsyncBatch(input1);
  // OK

  int input2 = getInput2();
  startAsyncBatch(input2);
  // OK

  int input3 = buggyGetInput();
  startAsyncBatch(input3);
  // bad implementation, gets bad input and throws
}
```

During execution, one error pops up, saying that the asynchronous batch received a bad input. It only indicates that somewhere in the code, `startAsyncBatch(int)` was called with an invalid argument. Imagine the stack trace going back before asynchronous execution. Finding the bug's origin is now a triviality.

As illustration, consider this test code.

```
public void test() {
  pathStep1();
}

private void pathStep1() {
  pathStep2();
}

private void pathStep2() {
  asyncExecutor1.execute(() -> pathStep3());
}

private void pathStep3() {
  pathStep4();
}

private void pathStep4() {
  asyncExecutor2.execute(() -> throw new TestError());
}
```

With standard stack traces, the error only indicates something like:

```
TestError: Unhandled intended testing error
        at Test.throwError(Test.java:102)
```

But using long stack traces produces:

```
TestError: Unhandled intended testing error
        at Test.throwError(Test.java:102)
```

```
   ****************************
   **        ASYNC GAP       **
   ****************************
at Test.asyncThrow(Test.java:98)
at Test.pathStep4(Test.java:94)
at Test.pathStep3(Test.java:90)
   ****************************
   **        ASYNC GAP       **
   ****************************
at Test.asyncExec(Test.java:86)
at Test.pathStep2(Test.java:82)
at Test.pathStep1(Test.java:78)
at Test.test(Test.java:158)
```

To implement this long stack trace, one could attribute a contextual value to each asynchronous execution containing the stack trace at the submission of the asynchronous code. Then, on error handling, simply recover the asynchronous stack trace and add it to the caught error.

2.4 Dependencies Tracking

When asynchronous programs gets large, ordering and dependencies among tasks can be the source of complex bugs. To solve them, we cruelly lack tools and representations to visualize how an execution did really happen. This can be done at a very fine level, collecting the exact time at which each task gets started and finished. A more useful approach is to collect only events where a task uses output of another task and create a dependency graph of that data.

Naively implementing such collection would be very cumbersome: each asynchronous task needs modification. Missing one of them results in a bug. Modifying each task results in code pollution and duplication. Trying to simplify the implementation by hooking underlying structures (for instance Java threads) does not work, since no one directly uses threads in his application, but rely on more efficient executors, as thread pools. Threads are only dumb workers and have not the ability to look at different tasks they execute.

But a tool that archives to automatically bind each asynchronous code to a context and capture transitions between these contexts can collect dependencies among asynchronous tasks and export a graphical representation, under the form of a directed graph for example. This solution is detailed in chapter 7.

2.5 Task Profiling

There are plenty of features we can imagine to ease the understanding and analysis of asynchronous programs. For example, one may be interested in how many total time a job required to execute (by opposition to how many time elapsed between the beginning and the end of that job).

```
public void test() {

  ParallelTask myTask = new ParallelTask();

  long start = System.currentTimeMillis();

  myTask.start();
  myTask.join();

  long end = System.currentTimeMillis();
  long elasedTime = end - start;
  // does not consider parallel sub-tasks of myTask
  // elapsed time is not the total execution time
}
```

Using right tools and abstractions for asynchronous tracing can make it easy to implement such profiling. In this case, a context that defines the “timer code” in the form of a hook around asynchronous execution is one solution. Then one simply has to wait for complete execution of all sub-tasks, collect all individual times from all sub-contexts and sum them to get the precise result.

Asynchronous Approaches

My work specifically targets the asynchronous world. Hence, I will start with a little tour of the most common asynchronous execution frameworks. Clearly, this vision is axed toward the Java programming language, but this is adapted for the current work, proposing a Java solution.

There is nothing to be done before a clarification of what I mean by "asynchronous". As many other buzz-words (see "reactive" for example), "asynchronous" is used in variety of situations for different meanings. Often, it describes a method call on a dedicated asynchronous API, with potential callback code that gets executed after completion.

Here, it describes more generally any code that does not run *in sequence* of the code that submitted it. It may run later if there is only one execution unit (single-threaded). It may run in parallel or later, if there is two or more execution unit (multi-threaded).

This can occur upon starting a new thread:

```java
(new Thread(() -> {
  // async task's code...
}).start();
```

Submitting a runnable to an executor:

```java
Executor executor = ...
executor.execute(() -> {
  // async task's code...
});
```

Registering a listener or event handler:

```java
class Listener implements MouseMotionListener {
  public void mouseMoved(MouseEvent e) {
    // async code
    System.out.println("Moved !");
  }

  public void mouseDragged(MouseEvent e) {
    // async code
    System.out.println("Dragged !");
  }
}

JPanel jPanel = ...

// async code submission
jPanel.addMouseMotionListener(new Listener());
```

Or any other way, as long as the submitted code does not execute in sequence of its submission.

Let's take a deeper look on the different approaches I analyzed before designing the Zone for Java.

3.1 Threads

In nowadays machines, threads are the basic unit of asynchronous execution. They are a very good tool, since they allow automatic time slicing via preemptive scheduling. This make them suitable for operative system, since no executed program has to be aware of interruptions, at the cost of context switching overhead.

The Java virtual machine ports this functionality to the programmer through its threads. However, the virtual machine does not allow context switching between those threads. Actually, the operating system supporting the Java threads transparently undertakes this role. Since one Java thread cannot be interrupted to change the task it is running (i.e. accomplish a context-switch), using Java's native threads is limited to one task per thread. Running a new task means instantiating and garbage-collecting a whole new thread.

3.2 Thread Pools

The high cost of allocating and collecting a Java thread is not affordable to reach high performances. This is why thread pools are very popular in Java. Rather than submitting the code to one thread, one submits it to the thread pool. When a thread is idle, the thread pool assign it a pending task, until no tasks are pending. The most recent Java implementation is the `ForkJoinPool`, bringing fine-tuned environment to execute parallel fork-join tasks.

In fact, there is an analogy between the single thread approach, relying on operating system scheduling and the thread pool. The thread pool acts as a non-preemptive scheduler for the Java virtual machine, and the pooled threads are its processes.

Thread pools bring notable speedup, avoiding costly thread instantiation, but loose the preemptive scheduling, with the risk of task starvation. And final consequence: one-to-one pairing between executed tasks and threads is lost. Threads become blind workers, with no control on the tasks they execute.

3.3 Callbacks

Callbacks are always associated with an asynchronous method. Usually, a callback is a piece of code (function pointer, anonymous class, lambda) taking as argument the output of its associated asynchronous method. The callback is not an asynchronous execution framework itself, but a way to append code to the asynchronous method. However, the callback still *declares* asynchronous code. Indeed, the callback code is submitted on method call and executed after the asynchronous method has returned. Note that the callback is not necessarily executed in sequence of its associated method. It can be asynchronous relative to that method too.

Callbacks are not mutually exclusive with thread pools. On the contrary, asynchronous methods and callbacks are positions in the program where the execution service can safely switch between different programs. You may have noticed the similarity with a program yielding the processor in a non-preemptive scheduler. Using asynchronous methods and callbacks breaks a task into smaller pieces indicating to the scheduler where it can safely pause an execution to undertake another sub-task. In this context, asynchronous methods typically allow to pause the execution of a blocking call, for full benefit. This explains why callback use is so important in JavaScript. Its non-preemptive single threaded executor needs many point to switch tasks and to archive effective parallelism.

The callback approach is less seen in Java, but is central in Dart and JavaScript. Recently, the popular Vert.x[1] framework successfully used this pattern on the Java virtual machine and illustrates how well the callbacks works with thread pools. Its API presents asynchronous methods with callbacks, but under the hood, relies on thread pools to get asynchrony.

3.4 Promises

An asynchronous method with a callback is a way to bind two codes together. The method runs, returns and then the callback executes. This binding is limited. Whether to bind one method call to multiple callbacks or reversely multiple method calls to one callback is not easy.

The promise is an answer to this limitation. It represents the future result of the asynchronous call. Its most interesting feature here is to bind code to execute after the asynchronous method has completed. This capacity of submitting code that executes later in time lets the promise enter the current asynchronous definition.

Called `CompletableFuture` in Java, they are present since Java 8 in the core library.

[1]Vert.x is a lightweight event-driven and non-blocking framework to build concurrent applications.

3.5 Observers

Speaking of Java and asynchronous programming, one cannot avoid the observer pattern. At the risk of oversimplifying the reality, I regroup in this category frameworks as Swing, based on observers or handlers and Akka[2] or Vert.x (again) who use the concept of actor, respectively verticle. Basically, an actor is an intelligent observer designed to have more control on the message it receives and to send messages to other actors. Regarding the asynchronous behavior, observers and actor do the same. They differ in term of concurrency control, but this is not the focus of this work.

Since observers get submitted, hold code executed later, they are also a valid case of an asynchronous code.

[2]Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.

The Zone

The Zone is an execution context associated to a code block. It represents the asynchronous extension of a scope. Like a scope, the Zone can contain other Zones.

The primordial purpose of Zones is to provide uniformity across asynchronous executions. It specifies constant key-value bindings accessible anywhere in the context (Zone values), task to execute when entering (cross-in hook) or exiting (cross-out hook) a Zone and hooks around synchronous (internal hook) and asynchronous (asynchronous hook) task submission. Those around hooks make it possible to manipulate the tasks that gets executed inside the Zone. While they are not essential to the concept of Zone itself, the opportunity to implement them naturally appears in a Zone implementation since the realization requires asynchronous submission hooks (see section 4.6). The Zone can then simply make these hooks accessible on its interface.

4.1 Model

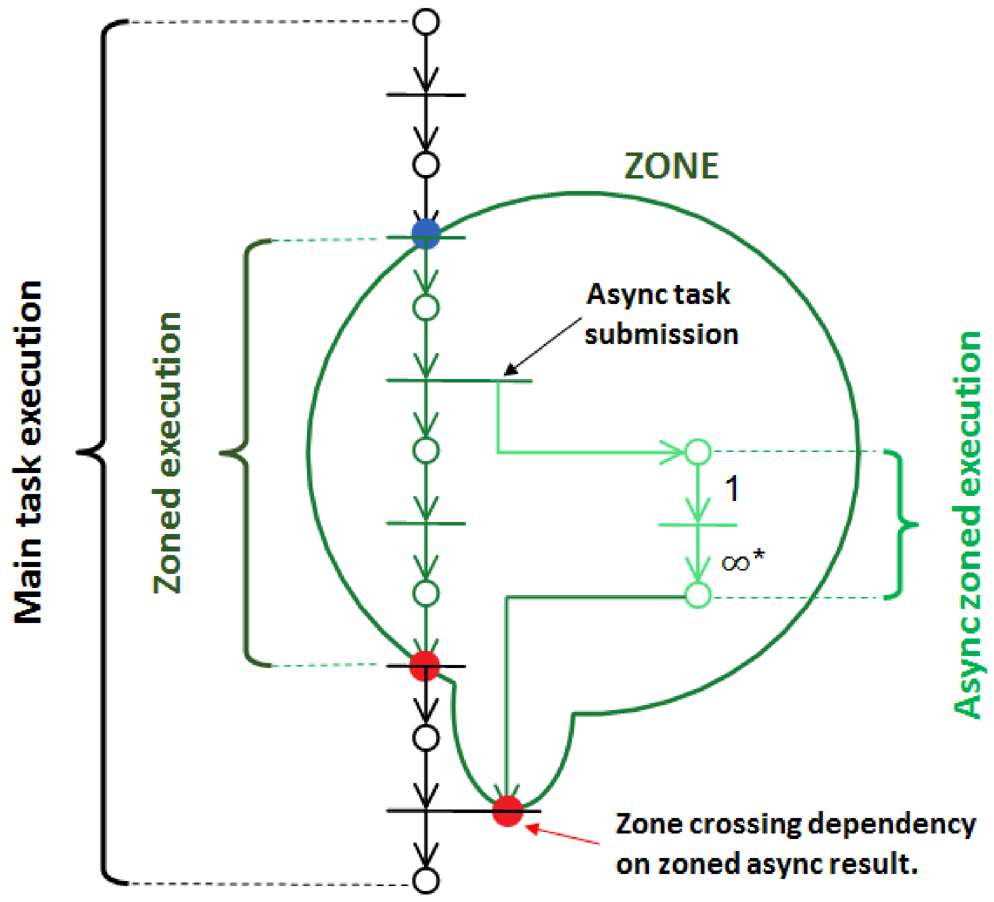
As illustrated in figure 4.1, the Zone can be understood as an extension to Petri net that allows to capture the tokens flow.

More formally, given a Petri net $PN = (P, T, F)$ with P the set of places, T the set of transitions and F the set of flows from P to T or T to P: $F \subseteq (P \times T) \cup (T \times P)$, a Zone Z is a subset $Z \subseteq P$ such that:

$$\forall Z_1, Z_2 \exists Z_0 \text{ s.t. } Z_1 \subseteq Z_0 \wedge Z_2 \subseteq Z_0$$

$$\forall Z_1, Z_2 (Z_1 \cap Z_2 \neq \emptyset) \Rightarrow [(Z_1 \subseteq Z_2) \vee (Z_2 \subseteq Z_1)]$$

Which more simply means: there exists a *root* Zone enclosing all Zones and Zones can contain other Zones, but cannot intersect otherwise (figure 4.2).



● Token crosses **inside** the **Zone**

● Token crosses **outside** the **Zone**

(*) ∞ symbolizes that the same asynchronous result may be used many times for different dependencies (the result is not "consumed").

- Apply **Cross-In** hook when token **enters** the **Zone**
- Apply **Cross-Out** hook when token **exits** the **Zone**
- Apply **Internal** hook to **Zoned Execution**
- Apply **Asynchronous** hook to **Async Zoned Execution**
- **Zone Values** are accessible anywhere inside the **ZONE**

Figure 4.1: Zone on Petri net

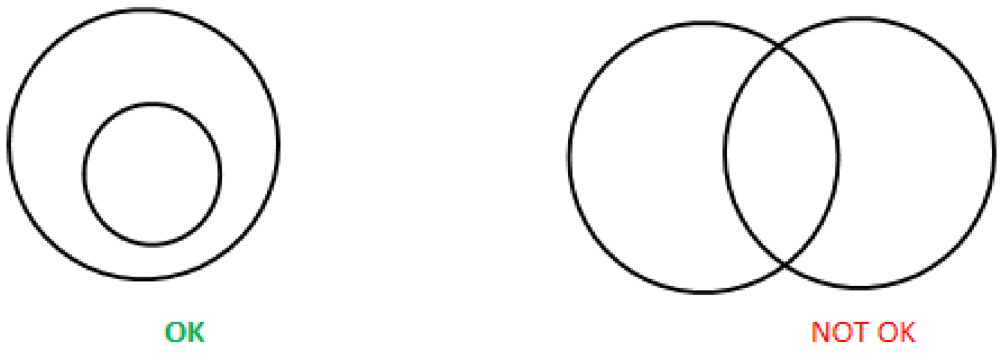


Figure 4.2: Zone intersection

The key-value bindings defined by the Zone are accessible from any place inside that Zone. The crossing hooks are triggered whenever a token crosses a Zone boundary.

The Zone identifies a crossing event when a token of the Petri net moves between two places that do not belong to the same Zones (see crossing algorithm in section 4.3). For more liberty in the crossing hooks, tokens may carry nothing (void tokens) an error (error tokens) or the result of a zoned task (result tokens). In Java, the tokens are implemented as simple objects containing a result, an error value or nothing.

In order to know which Zone originated a token, they are wrapped as zoned tokens. A zoned token is simply a tuple containing a token and its source Zone.

Operations on token while crossing Zone boundaries are described by the Zone's crossing hooks. The presence of around hooks defined in the Zone allows even more capabilities. However, unlike the crossing hooks, these around hooks are closer to an implementation feature than a primary effect of the model.

4.2 Terminology

In order to ease further discussion and presentation of the Zone, this section gives a small glossary of the essential terms used in the Zone.

Task represents the very general concept of “something the program has to do”. More formally, a task is a procedure with some input and zero or one output. For example, `Runnable` is a task with no input, no output. A method call is a task with method's arguments as input and method's return value as output.

Zoned Token is a tuple containing a token with its origin Zone. Zoned tokens are used as input to the crossing resolution algorithm (see section 4.3).

Zoned Task is the result of binding a task to a Zone. A Zoned task behaves like a task but accepts and return only zoned tokens. For instance a task expressed in Java by

```
Function<T, U> task;
```

Is associated to a zoned task represented by

```
Function<ZonedToken<T>, ZonedToken<U>> zonedTask;
```

Zone Stack is the stack of all enclosing zones at a given point of the program execution. Innermost Zone is considered to be at the top of the Zone Stack. If no Zone is defined, the current Zone is considered to be the default root Zone. For example :

```
(new ZoneA()).run(() -> {
  /*
   * Zone stack here is:
   * ZoneA, ROOT_Zone
   */
   (new ZoneB()).run(() -> {
     /*
      * Zone stack here is:
      * ZoneB, ZoneA, ROOT_ZONE
      */
   });
});
/*
 * Zone stack here is back to:
 * ROOT_ZONE
 */
```

4.3 Crossing Resolution

The crossing principle is the original feature of the Zone. The base mechanism is simple. Value outputted by zoned tasks are *zoned tokens*. When the value of a zoned token is used, it is extracted out of the zoned token to the current Zone, triggering the crossed Zones crossing hook.

Multiple cases can occur with the crossing mechanism. Especially in the Java implementation, where it is possible to keep and carry around references to already bound tasks, a token extraction can arise in unexpected contexts. Rather than relying on visibility restrictions to prevent such unusual cases to occur, the crossing resolution properly defines how the model behaves under such circumstances.

For better understanding, keep in mind that at the bottom of every single Zone stack, there is the Root Zone. The Root Zone can be seen as the Zone with no hooks and values enclosing the whole program.

In the given code examples, `new Zone()` creates a Zone whose parent is the enclosing Zone. For instance:

```java
Zone zone1 = new Zone();
Zone zone2;
zone1.run(() -> {
  // inside zone1
  zone2 = new Zone();
});

assert(zone1.getParent() == ROOT_ZONE);
assert(zone2.getParent() == zone1);
```

The crossing resolution algorithm always receives three parameters :

1. The source Zone
2. The destination Zone
3. The crossing Token

The crossing token is *not* a zoned token. The zoned tokens used by the framework provide the source Zone of the crossing resolution algorithm and the crossing token. The destination Zone is the current Zone of the extraction.

The crossing algorithm is divided in four steps:

1. Collect the Zone stack of the source Zone and the destination Zone to get the source stack and the destination stack.
2. Eliminate the common bottom part of the stacks. Recall that every Zone is a (indirect) child of the Root Zone, hence both stacks have at least one Zone in common. The idea of this step is to find the closest common enclosing the source and destination Zones: the *join Zone*.
3. Cross out the rest of the source stack in top-down order with the crossing token. This means calling each crossing hooks of the crossed Zone with the crossing token as argument.
4. Cross-in the rest of the destination stack in down-top order with the crossing token.

Let's see some concrete example of crossing resolutions.

Base case

The common case of zoned execution is running code in a child Zone.

```
// current Zone = some Zone

(new Zone()).run(() -> {
  // current Zone = new Zone

  // do zoned stuff here
});
```

When the code bound to "new Zone" starts, the crossing resolution algorithm executes with a very simple configuration, as described below.

Source Zone	Some Zone
Destination Zone	New Zone
Join Zone	Some Zone

Source Zone Stack	Some Zone :: ROOT_ZONE
Destination Zone Stack	New Zone :: Some Zone :: ROOT_ZONE
Common Part	Some Zone :: ROOT_ZONE

Pruned Source Stack	∅
Pruned Destination	New Zone

Since the join Zone is the source Zone, the left source stack after step 2 is empty. Step 3 of the algorithm has no effect. No cross-out hook is applied, only one cross-in, of the destination Zone, is executed.

When the code zoned in new Zone ends, the cross algorithm executes in the dual configuration, still simple:

Source Zone	New Zone
Destination Zone	Some Zone
Join Zone	Some Zone

Source Zone Stack	New Zone :: Some Zone :: ROOT_ZONE
Destination Zone Stack	Some Zone :: ROOT_ZONE
Common Part	Some Zone :: ROOT_ZONE

Pruned Source Stack	New Zone
Pruned Destination	∅

The join Zone remains the source Zone, but the left destination stack is emptied by step 2. Step 3 executes one cross out (from the source Zone), step 4 executes no cross-in. To summarize this case, a cross-in is executed when entering the new Zone, a cross-out is executed when entering the old Zone. So far, so good.

Running in parent Zone from child Zone

This case allows to see that starting code execution in a Zone does not necessarily implies crossing in.

```
// keeps reference to parent Zone
Zone parent = new Zone();

parent.run(() -> {
  // inside Zone parent
  Zone child = new Zone();
  child.run(() -> {
    // inside Zone child

    // this is the point of interest
    parent.run(() -> {
      // Zoned code here !!
    });
  });
});
```

When the code zoned in parent (at the point of interest) starts, the cross algorithm executes with the opposite configuration of the base case:

Source Zone	Child Zone
Destination Zone	Parent Zone
Join Zone	Parent Zone

When entering the Zone, the argument pattern is the same as when ending in the base case. Hence one cross-out will be executed, no cross-in. Reversely, one cross-in and no cross-out is executed when ending in the Zone. It is interesting to note that running in parent does not add layers on the Zone stack, but effectively pops the child Zone out of it. Adding once more the parent Zone on the Zone stack would not make sense, since the parent is an instantiation of a Zone, defining its own Zone stack (the context in which it was instantiated).

Entering a Sibling Zone

This last example shows how, in one single crossing resolution, can both cross-in and cross-out hooks be triggered.

```
(new Zone()).run(() -> {
  // inside parent Zone

  Zone child1 = new Zone();
  Zone child2 = new Zone();

  child1.run(() -> {
    // inside child1

    // this is the point of interest
    child2.run(() -> {
      // Zoned code here !!
    });
});
```

In this case, `child1` and `child2` share the same parent but neither is parent of the other one. Hence, when the code zoned in `child2` (at the point of interest) starts, the crossing resolution algorithm gets following configuration:

Source Zone	Child 1 Zone
Destination Zone	Child 2 Zone
Join Zone	Parent Zone

Source Zone Stack	Child 1 :: Parent Zone :: ROOT_ZONE
Destination Zone Stack	Child 2 :: Parent Zone :: ROOT_ZONE
Common Part	Parent Zone :: ROOT_ZONE

Pruned Source Stack	Child 1
Pruned Destination	Child 2

Unlike previous cases, the join Zone is neither the source nor the destination Zone. At the beginning of the zoned code, the step 3 triggers one cross-out from child 1, then the step 4 triggers one cross-in to child 2. As intended by the written code, at no point, parent Zone gets crossed in or out. The whole execution happens *inside* the parent Zone.

4.4 Task Binding

This section presents how Zones and tasks interact. Specifically, what makes a task *zoned* and how to bind a task to a Zone.

A zoned task has contextual access to the current Zone. At any point of its execution in a Zone, a task must be able to refer to that Zone. This reference gives access to the values of the local Zones.

The context of a zoned task persists asynchronously. If an asynchronous task is started from a Zone, it must keep contextual access to that Zone.

Every element crossing a Zone boundary triggers its *crossing hooks* (as discussed in section 4.3).

Code executed inside the Zone is intercepted and passed to its *internal hooks*. The code that executes inside a Zone can be manipulated by the Zone itself, according to its hooks specification. Here is an illustration of an internal Zone execution:

```
Zone myZone = ...

myZone.run(() -> {
  // code enclosed in this block is 'zoned' and
  // hooked as internal execution
  // by the 'internal hook' of myZone
});
```

In the same way internal execution is hooked, asynchronous submission in a zoned task is intercepted by the Zone. Asynchronous submission inside a Zone can be manipulated by the Zone itself, according to its hooks content. This typically occurs in this code pattern:

```
Zone myZone = ...

Executor myExecutor = ...

myZone.run(() -> {
  // inside myZone

  myExecutor.execute(() -> {
    // code enclosed here is still zoned in myZone and
    // hooked as asynchronous submission
    // by the 'async hook' of myZone
  });

});
```

Internal Binding

The internal binding and execution of a task follow six steps:

1. Task is bound to the Zone, internal hooks are applied.
2. The current Zone reference is updated to the Zone of execution.
3. An empty token crosses to the Zone of execution (crossing resolution applied).
4. The task, hooked by the Zone, executes.
5. The current Zone reference is updated to the initial Zone (before task execution).
6. The result token crosses to the initial Zone (crossing resolution applied).

Asynchronous Binding

The asynchronous binding and execution is quite different from the internal one. Unlike internally bound task, asynchronous task does not automatically trigger crossing operation. In fact, it is not possible to know a priori when the task will be executed, from which context will come its input (if the task depends on the execution of another task) neither in which context will be used its output (if another task depends on the execution of this task). This summarizes to the following two points:

- An asynchronously zoned task *input* may *come* from another Zone (hence causing crossing hook operations)

- An asynchronously zoned task *output* may *be used in* another Zone (hence causing crossing hook operations)

Since the crossing resolution is done on use-site, it takes into account its specific execution context:

- An input coming from the same Zone does not trigger any cross operation.
- An input coming from a parent Zone crosses inside the child Zone.
- An input coming from a child Zone crosses outside the parent Zone.
- An input coming from a sibling Zone (same parent) crosses outside the sibling Zone (towards the common parent Zone), then crosses inside the target Zone.

This implements the intuition that a Zone encloses everything happening inside it and the cross operations are triggered if and only if something enters or exits the Zone. For comparison with internal binding, asynchronous binding and execution of a task follow 6 steps:

1. Task is bound to the Zone, asynchronous hooks are applied.

   execution continues asynchronously...

2. The current Zone reference is updated to the Zone of execution.
3. All inputs cross to the Zone of execution (crossing resolution applied once per input).
4. The task, previously modified by the Zone's asynchronous hook, executes.
5. The current Zone reference is updated to the initial Zone.
6. The result is returned in a zoned token, bound to the emitting Zone. The crossing resolution is not applied immediately but later, each time the zoned result gets extracted.

4.5 Properties

The three aforementioned properties, crossing hooks, around hooks and Zone values are defined and used by the Zone as follows.

Zone Values

Zone values storage implements Java-like block scoping rules: an inner Zone sees and can shadow its outer Zone values, but the opposite does not hold. Since Zone values are accessible from anywhere (especially from concurrent contexts), the Zone storage is immutable to limit the risk of race conditions. Note that, in Java, it is still possible to have an immutable reference to a mutable value to bypass this restriction.

The Zone allows to lookup a key for one value, but also to fully lookup a key to retrieve the list of all values defined for that key in each Zone of the Zone stack.

Crossing Hooks

The crossing hooks are defined by the Zone as function from token to token. Note that in other code examples, the generic type parameter is omitted for the sake of simplicity.

```
public <T> Token<T> crossIn(Token<T>);
public <T> Token<T> crossOut(Token<T>);
```

Hence, the crossing hooks can inspect the token, make computations and return a *new* token with updated value. It is required that tokens are immutable. The reason is that the same zoned token may be used multiple times. If multiple tasks depend on the same asynchronous execution (hence having multiple callbacks for one method), its resulting token will be used once per dependency. Then it is crucial to always use the same unmodified token, not mentioning the risk of race conditions, if two concurrent crossing hooks run with the same zoned token.

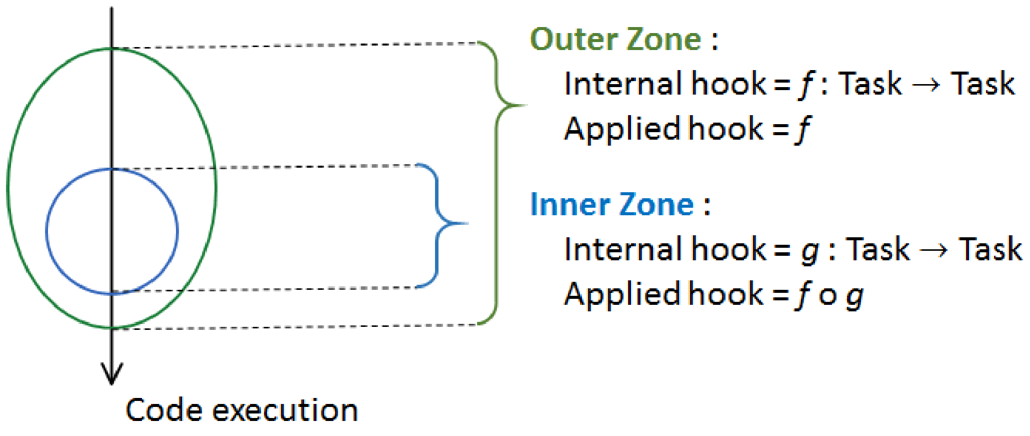
Around Hooks

Around hook discussion distinguished the internal hooks and asynchronous hooks, but both make the same task. Only their application locations differ. That is why they are not distinguished here and aggregated as "around hook".

Around hooks are represented as function from zoned task to zoned task. Again, for simplicity, other code examples do not show the generic type parameters.

```
public <T, U> ZonedTask<T, U> asyncHook(
        ZonedTask<T, U> zTask);
public <T, U> ZonedTask<T, U> internHook(
        ZonedTask<T, U> zTask);
```

This gives the opportunity to surround hooked task with a try-catch block, offers the choice to execute or not the hooked task and even to run the task in a dynamically created Zone. Moreover, in a Zone, around hooks are inherited from enclosing Zones and applied from innermost to outermost hook (figure 4.3).



Outer Zone :
Internal hook = f : Task → Task
Applied hook = f
Inner Zone :
Internal hook = g : Task → Task
Applied hook = $f \circ g$
Code execution

Figure 4.3: Around hook

Around Hooks Inheritance

For more flexibility, around hooks are inherited but not repeated. Consider three zones:

Zone	**Enclosing Zone**	**Defined hook**	**Applied hook**
outer Zone	None	f: Task $\rightarrow$ Task	f
middle Zone	outer Zone	g: Task $\rightarrow$ Task	$f \circ g$
inner Zone	middle Zone	f: Task $\rightarrow$ Task	$f \circ g$

In inner Zone, f hook does not get applied twice because it would repeat inherited f hook from outer Zone. When the same hook is inherited multiple times, the outermost origin determines the application order. In the previous example, inner Zone applies $(f \circ g)$ and not $(g \circ f)$ because the outermost origin of f (outer Zone) encloses the outermost origin of g (middle Zone). This prevents inner Zones from accidentally modifying inherited hooks application order.

This inheritance policy is more general than a blind straight-forward application of all hooks. It is easy (especially in Java) to re-create a different instance of a hook with the same functionality to prevent it from being filtered out. On the other hand, given a single function composed of many hooks, it is hard, if not impossible, to filter out repetitions to get such union-inheritance.

4.6 Realization

The hypothesis that any submitted asynchronous code can be hooked permits to safely ignore the Zone integration when discussing its functionalities. The integration will be always possible. This is a great benefit, since it effectively decouples the Zone functional specification from its integration concerns. The implementation is not depending on the multitude of evolving asynchronous frameworks in Java, as mentionned in chapter 3.

To integrate a Zone in a program, one can see it as a generalization of a Java thread local[1]. Note that the concept of thread local is not limited to Java.

[1]A Java thread local variable has an independently initialized unique copy for each thread

A mapping from threads to stored values implements the same behavior.

When using a thread pool, the concept of thread local becomes mitigated, even erroneous. Thread locals are bound to a specific thread, but the user does not controls how its tasks are bound to the threads. The common procedure to solve this is to wrap submitted code with instructions that update the thread local for the code.

```java
ThreadLocal<T> threadLocal = ...
Executor executor = ...

/*
 * Submitted code will update
 * the thread local on execution.
 */
public void submitWithLocal(Runnable code, T local) {
  Runnable newCode = () -> {
    // saves initial value
    T initial = threadLocal.get();

    // sets desired value for code
    threadLocal.set(local);
    code.run();

    // restores initial value
    threadLocal.set(initial);
  };

  executor.execute(newCode);
}
```

The Zone is a generalization of this method. Rather than binding a runnable, any code is accepted, denoted as *task*. The Zone handles internally the thread local to store its reference and simply exports a `bind` method. The Zone takes advantage of this binding operation to apply its internal mechanics and all defined around hooks. Based on this single binding, it provides:

- Asynchronous persistent context.
- Modeled execution flow between the Zones.

- Flexible programmable hooks on Zone transitions and code submission to the Zone.
- Extension and reusability mechanisms to extend and implement more Zone features.

This pattern efficiently decouples the Zone integration from its functionalities. To use it, one only needs to bind each submitted asynchronous task to the Zone. This can be even simpler using a “Zone-aware executor” (see chapter 7 for an example).

Chapter 3 showed that in Java, one cannot assume a single execution framework. The presented binding solution is not a uniform solution that automagically works in any context. It describes a single operation to execute on every asynchronous submission, which can even be modularized in the execution framework. That is how the Zone gets integrated in programs.

Applications

So far, the presented Zone is mostly theoretical. Around and crossing hooks have a precisely defined behavior, but beside Zone local storage, no feature answers the motivations for the Zone. I show here how to combine the Zone properties to build solutions to the initial problems.

5.1 Asynchronous Error Handling

Asynchronous error handling exists in two forms, contextual and systematic.

The contextual error handling consists in using the Zone as a standard try-catch block. The figure 5.1 illustrates the parallel between synchronous and contextual error handling.

```
// begin of async try-catch block
(new ErrorZone()).run(() ->

  future = asyncCall();

  // (1)
  String message = future.get();
  System.out.println(message);
);
// end of async try-catch block

// (2)
String message = future.get();
System.out.println(message);
```

```
// begin of sync try-catch bloc
try

  message = syncCall();

  // (1)
  System.out.println(message);

catch (Error e)  ...
// end of sync try-catch block

// (2)

System.out.println(message);
```

Figure 5.1: Asynchronous and synchronous try-catch block

In this example, `future = asyncCall()` starts an asynchronous task and returns the `Future` result of that task. For this exercise, the task will throw an error. Two cases are considered.

1. Access `message` inside the `ErrorZone`.
   The error is not handled by the `ErrorZone`. Nothing gets printed, since `future.get()` synchronously throws its error inside the `ErrorZone`.

2. Access `message` outside the `ErrorZone`.
   Upon `future.get()`, the result, zoned inside `ErrorZone` must cross out the Zone. The error gets handled and a fall-back value is provided.

The `ErrorZone` simply has to implement the crossing hook:

```java
/*
 * Checks if crossing token 'token' is an error token.
 * If yes, handle the error.
 */
public Token crossOut(Token token) {
  if (token.isErrorToken()) {
    Token fallback;
    // error handling ...
    return fallback;
  } else {
    // do nothing
    return token;
  }
}
```

The crossing resolution algorithm then accomplishes the work of selecting correct hooks according to the calling context.

Systematic error handling is an automation to uniformly wrap asynchronous code *inside* a try-catch block. This can be compared to Java's `Thread.UncaughtExceptionHandler` with the differences:

- The code catches the errors *before* some third-party execution service (a thread pool for example), preventing the execution to reach the `Thread.UncaughtExceptionHandler`.
- The setting is not global but local to the zoned portion of the code. With `Thread.UncaughtExceptionHandler`, one has to carefully set and unset the value.

- Many different default exception handler can be easily emitted by the same thread.

Such behavior is provided by a Zone implementing the asynchronous hook:

```
ZonedTask asyncHook(ZonedTask task) {
  return (input) -> {
    ZonedToken result = task.apply(input);
    if (result.isErrorToken()) {
      ZonedToken fallback;
      //handle logic, compute fallbackValue
      return fallback;
    } else {
      return result;
    }
  };
}
```

Since around hooks completely apply before crossing out the Zone, cross-out based error handling does not interfere with this code.

5.2 One Zone Per Asynchronous Task

This is not a direct solution to one problem of chapter 2, but an intermediate tool to build other features. The idea is to use an asynchronous hook to bind submitted task inside a new Zone.

```
ZonedTask aroundSubmit(ZonedTask task) {
  // creates a new instance of a custom Zone extension
  Zone newZone = new MyZone();
  ZonedTask newTask = newZone.bind(task);
  return newTask;
}
```

`MyZone` can be any Zone implementation, with the single constraint that it should override the internal hook application mechanism to do nothing. Otherwise, `newTask` may trigger unexpected internal hooks, inherited by enclosing Zones. The problem is that any *internal* hooks will be also applied where *asynchronous* hooks are.

5.3 Long Stack Traces

Implementation of long stack traces use one Zone per asynchronous task to set a Zone value containing the call stack trace of the asynchronous task. With contextual error handling, it simply defines an asynchronous hook to run submitted code inside a new Zone that:

- Stores as Zone local value the stack trace at the asynchronous submission.
- Wraps internal code inside a try-catch block to catch any error, extend it with the stored stack trace and re-throw it.

Long stack traces are also possible with systematic error handling, but require heavier mechanisms to ensure that the stack trace is always updated before being caught by the `catch` clauses of the error handling.

5.4 Asynchronous Dependency Tracing

Chapter 7 discusses in details asynchronous dependency tracking as a concrete experiment. The principle is however simple. Use one Zone per asynchronous task with IDs and implement a crossing hook to store the transitions from one Zone to another one to some data structure of your choice. One Zone per task creates a one-to-one correspondence between asynchronous tasks and IDs, the crossing hook detects and stores execution dependencies between two IDs.

Existing Approaches

The Zone exists already for Dart and JavaScript. Those frameworks were the inspiration of asynchronous execution context for the Java Zone. Both zone.js and dart:Zone define Zone local storage, execution hooks (in the sense of around hook for the Java Zone) and error handling. The error handling includes for both long stack trace generation. zone.js implements the systematic error handling for its call-back centered structure. dart:Zone follows the contextual error handling in its future-based approach.

	zone.js	**dart:Zone**	**Java Zone**
Multi threaded	no	no	yes
Zone local storage	yes	yes	yes
Distinction internal-asynchronous hook	yes	no	yes
***Around* hooks**	no	yes	yes
Extensible API	no	no	yes
Before, Return, After hooks	yes	no	buildable
Long stack trace	yes	yes	buildable
Promises support	no	yes	buildable
Crossing mechanism	no	no	yes
Out of the box basic tools	yes	yes	no
Adaptable to multiple frameworks	no	no	yes

Figure 6.1: zone.js, dart:Zone and Java Zone comparison

In comparison to zone.js and dart:Zone, the Zone for Java archives greater generality and, in my honest opinion, better founded execution principle. The Zone model gives arguments and a representation to understand how the Java Zone works, by opposition to zone.js and dart:Zone who settle for implementing use cases in answer to concrete problems.

Of course, this level of abstraction comes at the cost of increased complexity. For example, to get a long stack trace, one has to make a decision of how he will implement his asynchronous error handling: contextual or systematic? Then based on his decision, he chooses the correct hooks to add the asynchronous stack trace of the Zone into handled errors.

There is an emerging project, driven by StrongLoop, to provide a Zone in the Node.js environment. However it is not an adaptation of zone.js but a new implementation of the Zone, specific for Node.js. In the equivalent situation (see chapter 7), the integration of the Java Zone into Vert.x was almost transparent and required only to develop adapter code. No Zone functionality were rewritten and all existing specified Zone implementations are directly reusable.

This adaptability of the Java Zone, and its high composability are the main benefits brought over zone.js and dart:Zone.

Practical Experiments

This chapter shows two experiments of an advanced use of the Zone and their results. The first realizes an asynchronous sequence diagram of a parallel program with dependencies. The second describes how to transparently instrument the third-party framework Vert.x to realize a “Zone-aware Vert.x”.

7.1 Asynchronous Sequence Diagram

Specifying what is interesting in an asynchronous sequence diagram is the most complex part if this experiment. In fact, producing the information of a conventional sequence diagram is not interesting. There are already many tools to do that. The useful data is an asynchronous dependency diagram: what are the dependencies between asynchronous tasks. The produced diagram is defined by:

- Each task execution is divided into sub parts, represented by the nodes of a graph.
- Tasks are divided into the biggest possible parts such that a sub-part execution first uniquely shows *incoming* dependencies (rely on other tasks result or completion). Then it uniquely shows *outgoing* dependencies (another task shows incoming dependency from this task).
- Dependencies are represented as directed edges between the corresponding nodes of their tasks.

The produced scheme is a directed acyclic graph where each node represents an autonomous part of the programs and the edges show the precedencies between those parts.

The realization using the Zone is concise. As evoked in chapter 5, I use the one Zone per task pattern, creating `TaskZones`. Each `TaskZone` stores a unique task ID and defines the crossing hook:

```
// The cross operation executes inside -destination- Zone

// The cross operation is defined inside -source- Zone

public Token crossOut(Token token) {
  TaskZone sourceZone = this;
  int sourceID = sourceZone.taskId;

  int destID = Zone.lookup(TASK_ID_KEY);

  storeDependency(souceID, destID);
}
```

The longest part is then to compile collected information and build a graphical representation. For the code of a fork-join algorithm (figure 7.1) I used a sample code from the d3.js library to produce the view shown in figure 7.2. Each node is named 'X#Y'. 'X' is the task id and 'Y' the subtask id. The initial task is '0', split in two subtasks '0#1', entry point of the algorithm, and '0#2', end of the algorithm with output value.

Solving problem P

1. If P is small then solve P.
2. Else split P in a list Ps of smaller problems p.
3. For each p in Ps, start an asynchronous task to solve p.
4. Join each asynchronous task started in (3) and collect all sub-results.
5. Compose the sub-results and return the final solution.

Figure 7.1: Generic fork-join algorithm

7.2 Transparent Instrumentation

In this experiment, I used the AOP framework AspectJ to automatically instrument the Vert.x library. If analyzing the Vert.x code to define where and how to apply the Zone was tedious, the concrete realization was much more straightforward.

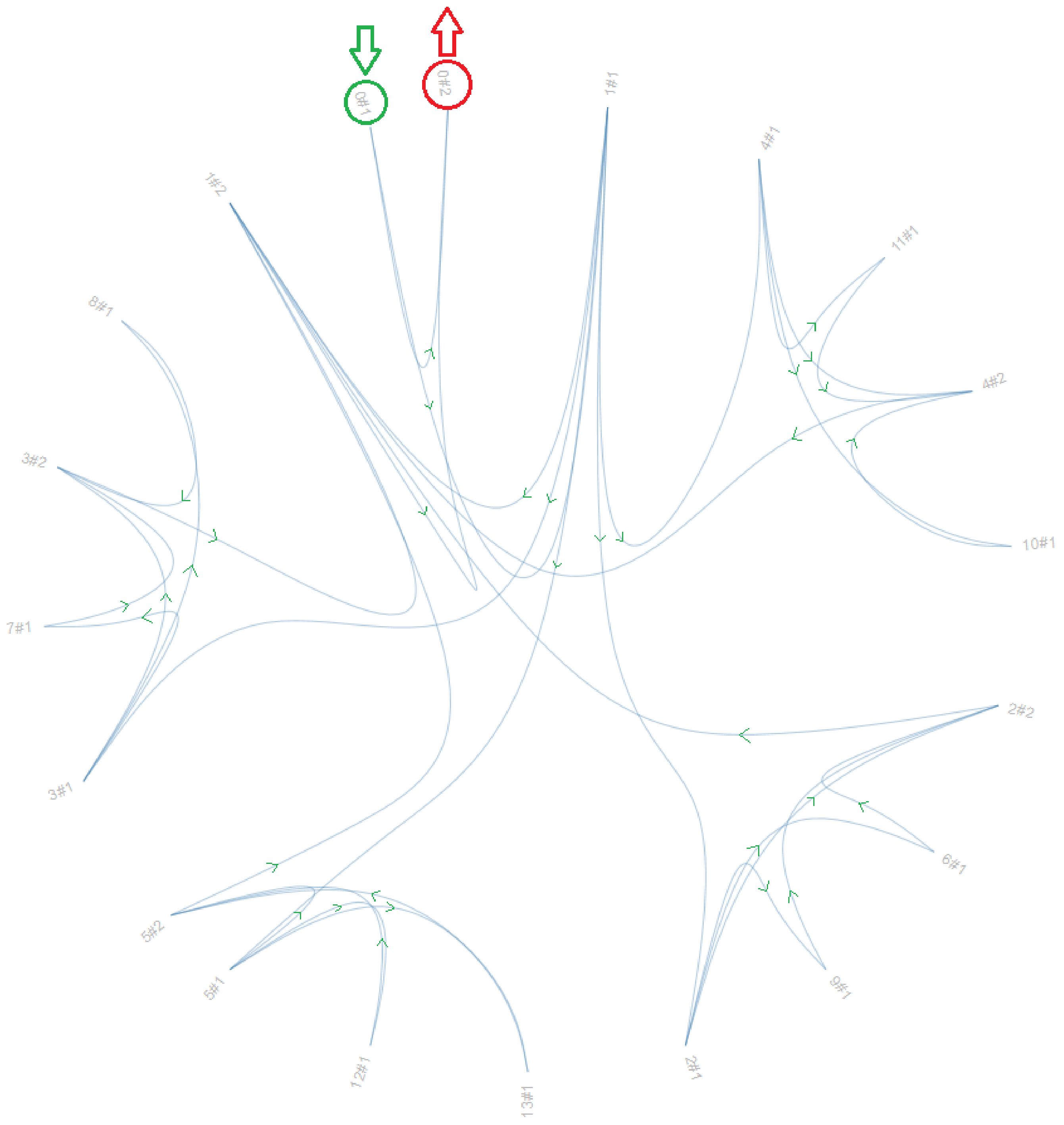


Figure 7.2: Fork-join algorithm dependency graph

Results of this work are:

- Integration code is really short (only six classes of 60 lines in average).
- Aspect weaving only relies on public Vert.x API and assumes no internal behavior for a greater stability.
- The instrumentation is done by a stand-alone program that produce an artifact interchangeable with the original Vert.x.

"Interchangeable" here means one can transparently be replaced by the other. If used on an existing project, the only thing to modify is the import resolution in the project configuration. Since the instrumented Vert.x library completely reproduces the original package structure, the imports in the source code do not need modification.

With this instrumented library, I was even able to reuse the classes developed for the asynchronous dependency diagram to visualize the dependencies of a Vert.x program.

Conclusion

We have seen that the Zone for Java brings innovative building tools. The integration work may be complex (as showed the Vert.x case study), but once this is correctly done, one can almost transparently rely on the Zone as a uniform execution context. In summary, the Zone needs:

- Explicit binding to asynchronous tasks submitted to the framework.

In return, it brings:

- Persistent context over asynchronous transitions.
- Modeled execution flow between Zones.
- Flexible programmatic hooks on:
  - Zone binding and execution.
  - Asynchronous submission and execution.
  - Execution flow across different Zones.
- Extensible and reusable structure.

To close this work, I'll show three ways of research and development on the Zone and a short analysis of how the original problematic changed through the research.

8.1 Development

Bringing more and easier binding primitives would be very interesting. Even though the binding principle is detailed, its implementation is still not general. This is mainly due to the large amount of standard interfaces present in the `java.util.function` package that match to the Zone's task abstraction. Actually, the Zone only handles `BiFunction` and requires additional wrap-steps to bind another interface, as `Runnable` for example.

When using and probing the Zone, I often encountered the need to ignore parts of the Zone stack, or even to reorder it. Reordering would be useful to ensure some hook is always applied in first position. The union-inheritance is the solution I used to gain the required flexibility. However, nothing proves this is the only solution.

The scope projection is one solution to theorize and specify the Zone stack manipulation. It consists in using a predicate on the Zone to filter the Zone stack, allowing to transform the Zone context in another one with only the properties of accepted Zones. One concrete use is to deduce a new context, based on the current one. This grants an easy way to keep some useful information and drop the data we don't use any more.

8.2 Theory

The representation on Petri nets is useful for understanding but not totally formalized. A strict theorizing will allow to relate, compare and even associate the Zone to existing Petri nets extensions. For example, the colored Petri nets have the ability to store data inside its token, as the Zone does with input and return values in zoned tokens. If such association exists, the Zone could benefit of existing results and asynchronous theorems, hence increasing its stability and the understanding we have of the Zone.

8.3 Practice

The zone stays a complex abstraction, especially realizing an integration on an execution framework is hard. A good improvement is to collect experience data and compile it as a best practices guide or a trouble shooting documentation. This can even lead to reworking some aspects of the model that would appear unsound.

An additional performance analysis will help to understand the cost of using the Zone. Indeed, it currently adds a lot of wrapping overhead. Knowing this overhead and the bottlenecks of the framework opens the door to efficient optimization and determines if it suits to a production environment.

8.4 Asynchronous problematic

It was interesting to see how the central problematic evolved in the course of the development. At first, I was focused on creating a working implementation with hooks, with error handling, with values, and all other requirements to solve the asynchronous problems. Quickly, I realized that I was only treating the symptoms of the problem and started to look for analogies of all these different solutions. Is there an essential problem and a unique solution, on which all other specific properties can be realized?

Trying to answer that question, I found out that the real need is a *uniform execution principle*, constant in both synchronous and asynchronous transitions. This gave birth to the crossing principle that I successfully used to implement specialized abilities on the Zone.

With these words, we can conclude that the real need to correctly deal with the asynchronous programming is not an overpowered framework with lots of features and an incredible refinement to distinguish addressed cases. The real point is to find a simple representation to think about synchronous and asynchronous executions in a *uniform way.*

References

[1] Erika Ábrahám, Immo Grabe, Andreas Grüner, Martin Steffen, *Behavioral interface description of an object-oriented language with futures and promises*, Elsevier, 2008.

[2] Dima Alhadidi, Nadia Belblidia, Mourad Debbabi, Prabir Bhattacharya, *λ_AOP: An AOP Extended Lambda-Calculus*, CSL-CIISE, Concordia University, 2007.

[3] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazires, Robert Morris, *Event-driven Programming for Robust Software*, MIT Laboratory for Computer Science, 2002.

[4] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea, *Java Concurrency in Practice*, Addison-Wesley, 2010.

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, *The Java® Language Specification (Java SE 8 Edition)*, Oracle America, 2014.

[6] Lauer H.C., Needham R.M., *On the Duality of Operating Systems Structures*, inProc. Second International Symposium on Operating Systems, 1978.

[7] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java® Virtual Machine Specification (Java SE 8 Edition)*, Oracle America, 2014.

[8] Ingo Maier, Martin Odersky, *Deprecating the Observer Pattern with Scala.React*, LAMP-EPFL, 2012.

[9] Joachim Niehren, Jan Schwinghammer, Gert Smolka, *A Concurrent Lambda Calculus with Futures*, Theoretical Computer Science, Elsevier, 2006.

[10] Meng Wang, Bruno C. d. S. Oliveira, *What Does Aspect-Oriented Programming Mean for Functional Programmers*, Oxford University Computing Laboratory, 2009.