7.7 Weitere Konzepte und Programmiermuster zur nebenläufigen Programmierung

Neben den bisher vorgestellten Konzepten gibt es noch viele weitere Ansätze bzw. Programmiermuster zur Programmierung nebenläufiger Systeme, von denen wir einige nachfolgend vorstellen.

7.7.1 Aktoren

Aktorbasierte Programmierung ist ein Strukturierungskonzept für nebenläufige Systeme, bei dem die einzelnen Komponenten unabhängig und parallel ablaufen. Obwohl das Aktor-Modell schon 1973 entwickelt wurde (Hewitt et al., 1973), wird es auch heute noch als Basismodell in verschiedenen Programmiersprachen verwendet oder kann auch in Form von Bibliotheken in Programmiersprachen genutzt werden.

Die Idee des Aktor-Modells ist, dass alle nebenläufigen Aktivitäten in Form von Aktoren repräsentiert werden. Ein **Aktor** ist dabei eine Berechnungskomponente, die mittels Nachrichtenaustausch mit anderen Komponenten kommuniziert. Da es keinen gemeinsamen Speicher zwischen verschiedenen Aktoren gibt, können die einzelnen Aktoren unabhängig und parallel arbeiten, d.h. es gibt keine kritischen Bereiche. Jeder Aktor arbeitet auf seinen eigenen privaten Daten. Jeder Aktor kann die folgenden Aktionen ausführen:

- Nachrichten empfangen und verarbeiten
- Nachrichten an andere Aktoren senden
- seinen lokalen Zustand verändern
- weitere Aktoren starten

Da Aktoren Nachrichten von beliebigen anderen Aktoren empfangen können, hat jeder Aktor auch eine Nachrichtenwarteschlange, in die eingehende Nachrichten eingereiht werden.

Das Aktor-Modell ist einerseits ein theoretisches Modell für nebenläufige Berechnungen und andererseits eine Basis für praktische Implementierungen. Zum Beispiel beruht das nebenläufige Programmierkonzept von Erlang auf dem Aktor-Modell. Weil Erlang eine funktionale Basissprache hat, wird der lokale Zustand einer Erlang-Prozesses durch die Parameter einer Funktion repräsentiert, d.h. die Zustandsänderung erfolgt durch einen rekursiven Aufruf mit veränderten Parametern. Die Nachrichtenwarteschlange ist nicht direkt zugreifbar, sondern wird durch das `receive`-Konstrukt abgefragt.

Für andere Programmiersprachen gibt es Bibliotheken, die eine Aktor-basierte Programmierung unterstützen. So unterstützt z.B. die Sprache Scala (vgl. Kapitel 5.6.1) Aktoren durch eine mitgelieferte Klassenbibliothek. Alternativ gibt es auch die Bibliothek Akka[2], die das Aktor-Modell auf der JVM unterstützt und in Java und Scala genutzt werden kann. Ebenso gibt es für die Sprachen C++ und Go[3] Implementierungen des Aktor-

[2] `https://akka.io`
[3] `https://golang.org`

Modells. Typisch für Implementierungen des Aktor-Modells ist die Leichtgewichtigkeit von Aktoren, d.h. Aktoren sind leichtgewichtige Prozesse, sodass man ohne Probleme zehntausende von Aktoren erzeugen kann.

7.7.2 Futures

Häufig möchte man bei nebenläufigen Berechnungen nicht nur Prozesse starten, sondern diese Prozesse sollen auch bestimmte Werte berechnen, auf die man dann später zugreift und weiter verwendet. Da man beim Starten eines Prozesses typischerweise nur einen Identifikator für den neuen Prozess erhält, muss man die Rückgabe von berechneten Werten explizit realisieren. Z.B. kann man bei Prozessen mit geteiltem Speichern (wie in Java) den Wert in einer gemeinsamen Variable bereitstellen, wobei man mittels Signalen mitteilen muss, wann dieser Wert bereitsteht. Bei nebenläufiger Programmierung mit Nachrichtenweitergabe (z.B. in Erlang) kann der neue Prozess den berechneten Wert mittels einer Nachricht senden.

Diese Lösungen erscheinen sehr umständlich und erfordern eine explizite Codierung der Wertrückgabe. Eine elegantere Lösung haben wir beim Rendezvous-Konzept von Ada kennengelernt: da das Rendezvous mittels Prozeduraufrufen erfolgt, kann man wie bei Funktionsprozeduren das Ergebnis einfach als Rückgabewert definieren. Das Rendezvous-Konzept hat allerdings den Nachteil, dass der Aufrufer so lange wartet, bis der Server den Aufruf vollständig abgearbeitet hat, was den Durchsatz paralleler Berechnungen verlangsamen kann.

Zur Lösung dieser Probleme wurde in den 1970er Jahren die Idee von *Futures* (oder auch *Promises*) entwickelt. Konzeptuell ist ein **Future** ein Objekt, dessen Wert z.B. von einem anderen Prozess noch berechnet wird. Insofern kann man ein Future in Ausdrücken verwenden, auch wenn der Wert noch nicht bereitsteht. Erst wenn der Wert tatsächlich benötigt wird, wird der Prozess, der den Wert benötigt, solange angehalten, bis der Wert tatsächlich bereit steht. Somit kann das Future-Konzept ähnlich wie das Rendezvous-Konzept benutzt werden, allerdings wartet der Aufrufer nicht, sondern er kann direkt weiterrechnen und wartet erst auf den Server, wenn er das Ergebnis benötigt.

Futures können also verwendet werden, um z.B. vorhandene sequenzielle Implementierungen zu parallelisieren, ohne dass die Struktur des Algorithmus verloren ght. Wenn dabei Teilergebnisse parallel unabhängig berechnet werden und deren Ergebnisse erst spät benötigt werden, erhält man durch diese Art der Parallelisierung eine gute Effizienzsteigerung.

Futures stehen in vielen Programmiersprachen (z.B. C++, Java, Scala) in Form von Bibliotheken zur Verfügung. Z.B. stellt Java (seit Version 5) im Paket `java.util.concurrent` das Interface `Future`[4] zur Repräsentation von Future-Objekten zur Verfügung. Diese enthält im wesentlichen die Methode `get()`, um auf den Wert des Future-Objektes zuzugreifen. Ein `FutureTask` ist eine Implementierung von `Future`, die auch die Schnittstelle `Runnable` implementiert, sodass diese auch in einem Thread ausgeführt werden kann. Bei

[4] `https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/Future.html`

der Definition eines `FutureTask` muss die Berechnung selbst als `Callable` z.B. mittels lambda-Notation definiert werden. Das funktionale Interface `Callable` besteht aus einer Methode:

```
public Interface Callable<V> {
  V call() throws Exception;
}
```

Ein `FutureTask` enthält somit ein `Callable`-Objekt, das nebenläufig berechnet und bei dessen Wertzugriff synchronisiert wird. Das folgende Beispiel zeigt eine mögliche Verwendung dieser Klasse:

```
...
Callable<String> world = () -> { return "World"; };
FutureTask<String> futureWorld = new FutureTask<String>(world);
new Thread(futureWorld).start();
...
System.out.println("Hello " + futureWorld.get() );
```

Der `FutureTask` enthält hier also die Berechnung `world`, die nebenläufig ausgeführt wird und dessen Ergebnis mit dem Aufruf `futureWorld.get()` abgefragt wird. Die Schnittstelle `Future` stellt noch weitere Methoden wie z.B. `get` mit Timeout und `cancel` zur Verfügung. `get` selbst kann auch verschiedene Ausnahmen werfen.