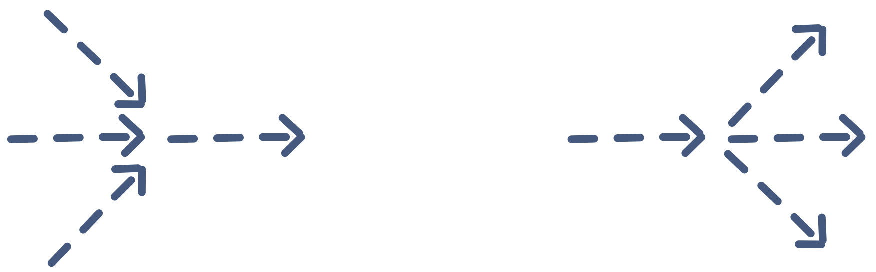
Fan-In, Fan-Out

In this lesson, we will get familiar with Fan-In, Fan-Out techniques which are used to multiplex and demultiplex data in Go.

Fan-In refers to a technique in which you join data from multiple inputs into a single entity. On the other hand, Fan-Out means to divide the data from a single source into multiple smaller chunks. In this lesson, we'll learn how to make use of both these techniques.



Fan-In Fan-Out

The code below is from the previous lesson where two receiving operations were blocking each other.

```
    fmt.Println(<-positionChannel1)
    fmt.Println(<-positionChannel2)
```

These operations were taking turns not only in printing value on to the console but also in proceeding to the next computation.

Check it out below:

```
package main

import ( "fmt"
                "math/rand"
                "time")

func updatePosition(name string) <-chan string {
        positionChannel := make(chan string)

        go func() {
                for i := 0; ; i++ {
                        positionChannel <- fmt.Sprintf("%s %d", name , i)
                        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
                }
        }()

        return positionChannel
}

func main() {
        positionChannel1 := updatePosition("Legolas :")
        positionChannel2 := updatePosition("Gandalf :")


        for i := 0; i < 5; i++ {
                fmt.Println(<-positionChannel1)
                fmt.Println(<-positionChannel2)
        }

        fmt.Println("Done with getting updates on positions.")
}
```

But what if you want to get position updates as soon as they are updated? This is where the fan-in technique comes into play. By using this technique, we'll combine the inputs from both channels and send them through a single channel. Look at the code below to see how it's done:

```
package main

import ( "fmt"
                  "math/rand"
                  "time")

func updatePosition(name string) <-chan string {
        positionChannel := make(chan string)

        go func() {
                for i := 0; ; i++ {
                        positionChannel <- fmt.Sprintf("%s %d", name , i)
                        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
                }
        }()

        return positionChannel
}

func fanIn(mychannel1, mychannel2 <-chan string) <-chan string {
        mychannel := make(chan string)

        go func() {
                for {
                        mychannel <- <-mychannel1
                }

        }()

        go func() {
                for {
                        mychannel <- <-mychannel2
                }
        }()

        return mychannel
}


func main() {
        positionsChannel := fanIn(updatePosition("Legolas :"), updatePosition("Gandalf :"))


        for i := 0; i < 10; i++ {
                fmt.Println(<-positionsChannel)
        }

        fmt.Println("Done with getting updates on positions.")
}
```

The crux of our technique lies in the `fanIn` function:

```
func fanIn(mychannel1, mychannel2 <-chan string) <-chan string {
    mychannel := make(chan string)

    go func() {
        for {
            mychannel <- <-mychannel1
        }
    }()

    go func() {
        for {
            mychannel <- <-mychannel2
        }
    }()

    return mychannel
}
```

Here we take in two channels as input and specify one channel as the return argument, i.e., `mychannel` which is the Fan-In channel. Afterward, we declare two goroutines from **line 23** to **line 33** which receive data from `mychannel1` and `mychannel2` and send it to our Fan-In channel, `mychannel` on **line 25** and on **line 31**. Hence, data will be passed to `mychannel` as soon as it is received by `mychannel1` and `mychannel2` because the goroutines are running concurrently.

Additionally, we pass the `updatePosition` function for both Legolas and Gandalf into our `fanIn` function on **line 40**. The rest of the logic is the same as before. The only difference comes from the fact that `mychannel1` and `mychannel2` are communicating with `mychannel` now instead of directly communicating with the main routine as before. You will realize from the output that the position updates are no longer sequential. Thus, by using this technique, we can solve the blocking issue that we were previously facing.

Let's jump to the Fan-Out technique now. The code below generates an array of random numbers and prints all the values after doubling them.

```
package main

import (
	"fmt"
	"math/rand"
	"strconv"
	"time"
)

func main() {
	var myNumbers [10]int
	for i := 0; i < 10; i++{
		rand.Seed(time.Now().UnixNano())
		myNumbers[i]=rand.Intn(50)
	}

	mychannelOut := channelGenerator(myNumbers)

	mychannel1 := double(mychannelOut)
	mychannel2 := double(mychannelOut)

	mychannelIn := fanIn(mychannel1, mychannel2)


	for i := 0; i < len(myNumbers); i++ {
		fmt.Println(<-mychannelIn)
	}
}

func channelGenerator(numbers [10]int) <-chan string {
	channel := make(chan string)
	go func() {
		for _, i := range numbers {
```

```
                                channel  <-  strconv.Itoa(i)
                        }
                        close(channel)

                }()
                return channel
}

func double(inputchannel <-chan string) <-chan string {
                channel := make(chan string)
                go func() {
                        for i := range inputchannel {
                                num, err := strconv.Atoi(i)
                                 if err != nil {

                                 }
                                 channel <- fmt.Sprintf("%d * 2 = %d", num,num*2)
                        }
                        close(channel)
                }()
                return channel
}


func fanIn(inputchannel1, inputchannel2 <-chan string) <-chan string {
                channel := make(chan string)
                go func() {
                        for {
                                select {
                                case message1 := <-inputchannel1:
                                        channel <- message1
                                case message2 := <-inputchannel2:
                                        channel <- message2
                                }
                        }
                }()
                return channel
}
```

Let’s analyze what’s happening above. From **line 12** to **line 15**, we are just populating an array with random numbers ranging from 0 to 49.

```
for i := 0; i < 10; i++{
          rand.Seed(time.Now().UnixNano())
          myNumbers[i]=rand.Intn(50)
}
```

On **line 17**, we create a common channel `mychannelOut` using the `channelGenerator` function.

The `channelGenerator` function is as follows:

```go
func channelGenerator(numbers [10]int) <-chan string {
    channel := make(chan string)

    go func() {
        for _, i := range numbers {
            channel  <-  strconv.Itoa(i)
        }
        close(channel)
    }()
    return channel
}
```

It returns a channel which will receive data from the goroutine created in the function itself. We are just converting the integer values from the input array to string and sending on to the channel.

In the main routine (**lines 19-20**), we *fan-out* our common channel `mychannel` to two goroutines, which are created from inside the function `double`.

```go
func double(inputchannel <-chan string) <-chan string {
    channel := make(chan string)
    go func() {
        for i := range inputchannel {
            num, err := strconv.Atoi(i)
             if err != nil {
                // handle error
             }
             channel <- fmt.Sprintf("%d * 2 = %d", num,num*2)
        }
        close(channel)
    }()
    return channel
}
```

The `double` function takes our common channel as an input and sends data on to another channel. Note that there are two instances of the `double` function, which implies that the two goroutines are concurrently receiving data from a single channel. This is pretty much our fan-out technique where we distribute the data sent from one channel to two channels concurrently, which divides the computation of doubling the integers and returns them as strings among the two goroutines.

On **line 22**, we fan-in the results from the channels returned by the `double` function and print them on to the console.

function and print them on to the console.

Isn’t that amazing? The Fan-In, Fan-Out techniques can be pretty useful when we have to divide work among jobs and then combine the results from those jobs.

Let’s move on to the next lesson in which we will restore sequencing in our code.