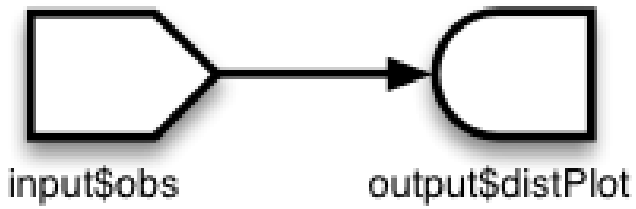
# Reactividad avanzada en Shiny

# Resumen de reactividad (Lectura reactividad I)

Link a la lectura

# ¿Qué es la reactividad?

```
server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
```



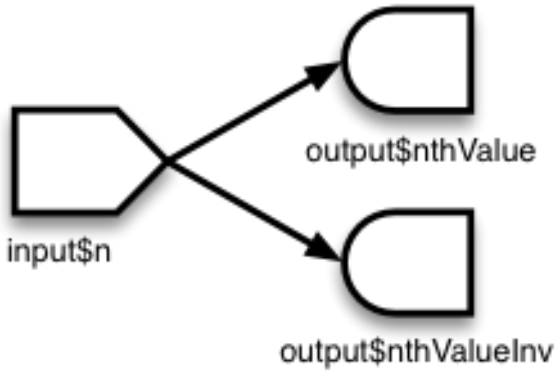

Al usar un input dentro de un output se crea esta relación de reactividad

Siempre es input -> output

# Relaciones 1:N o N:1

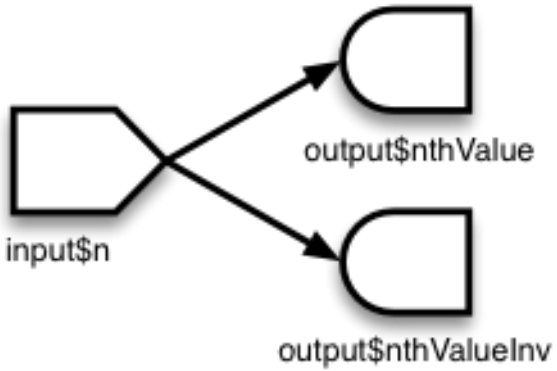
No tiene que ser una única flecha. Puede ser varios inputs para un output o un input para varios outpus.

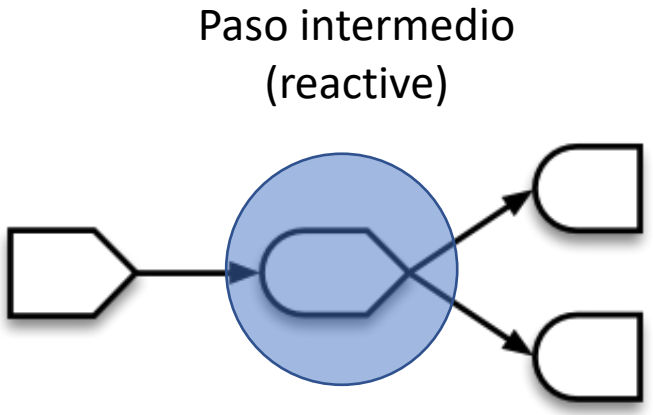
```r
# Calculate nth number in Fibonacci sequence
fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))

server <- function(input, output) {
  output$nthValue    <- renderText({ fib(as.numeric(input$n)) })
  output$nthValueInv <- renderText({ 1 / fib(as.numeric(input$n)) })
}
```



# Reactives (“valores intermedios”

A veces que un input afecte a dos outputs es ineficiente. Calculamos dos veces (o más) algo que puede ser muy pesado. Para ello hacemos un paso “intermedio”

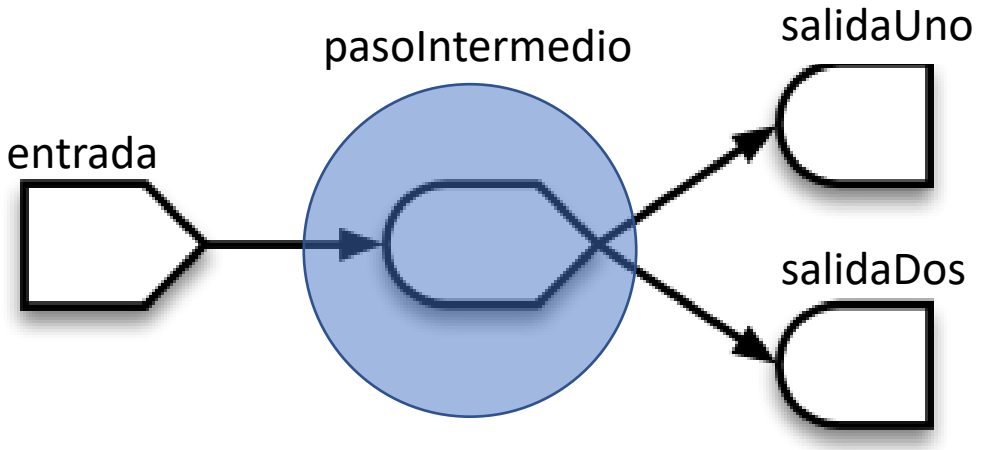


# Ejemplo de reactive: ahorrar trabajo

Este código se ejecuta sólo una vez por cada vez que entrada cambie entrada

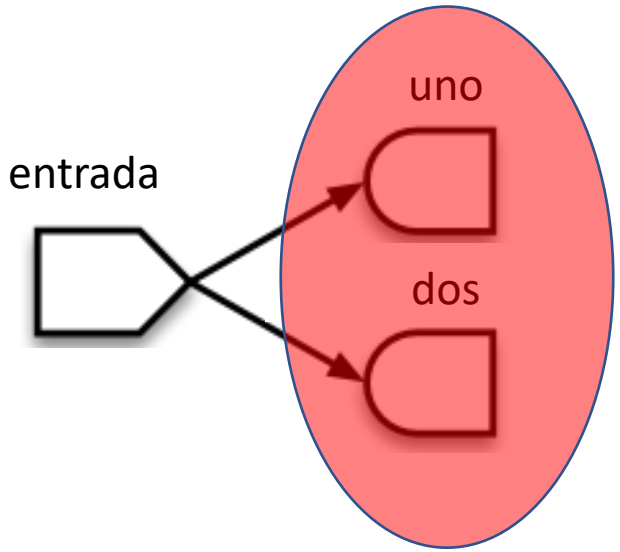
```
server <- function(input, output) {
  # Aquí hacemos el trabajo pesado para que sólo se haga una vez
  pasoIntermedio <- reactive({ trabajo_costoso(input$entrada) })

  # Todos los inputs que usen pasoIntermedio
  output$salidaUno  <- renderText({ pasoIntermedio() })
  output$salidaDos <- renderText({ pasoIntermedio() + 1 })
}
```



# Ejemplo SIN reactive (MAL hecho)

Cada output se ejecuta por separado ejecutando dos veces trabajo_costoso

```
server <- function(input, output) {
  output$uno <- renderText({ trabajo_costoso(input$entrada) })
  output$dos <- renderText({ trabajo_costoso(input$entrada) })
}
```

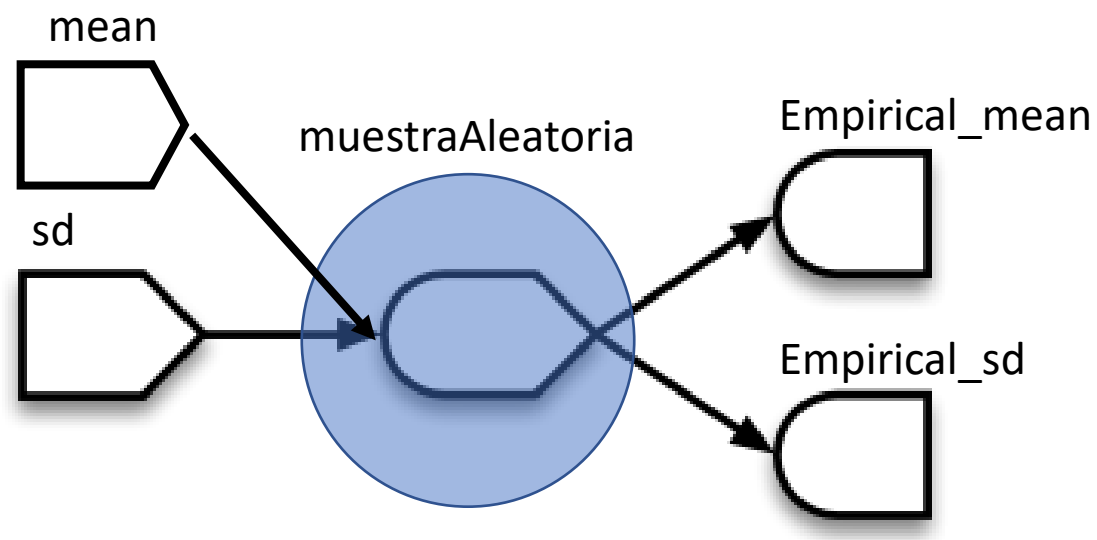


Contenido nuevo

# Ejemplo de reactive: procesos aleatorios

```
server <- function(input, output) {
  muestraAleatoria <- reactive({ rnorm(100, input$mean, input$sd) })

  output$empirical_mean <- renderText({ mean(muestraAleatoria()) })
  output$empirical_sd <- renderText({ sd(muestraAleatoria()) })
}
```

Usan la misma muestra aleatoria “a la vez”



Contenido nuevo

# Importante: sintaxis de reactive

Llaves (como en renderXXXX)

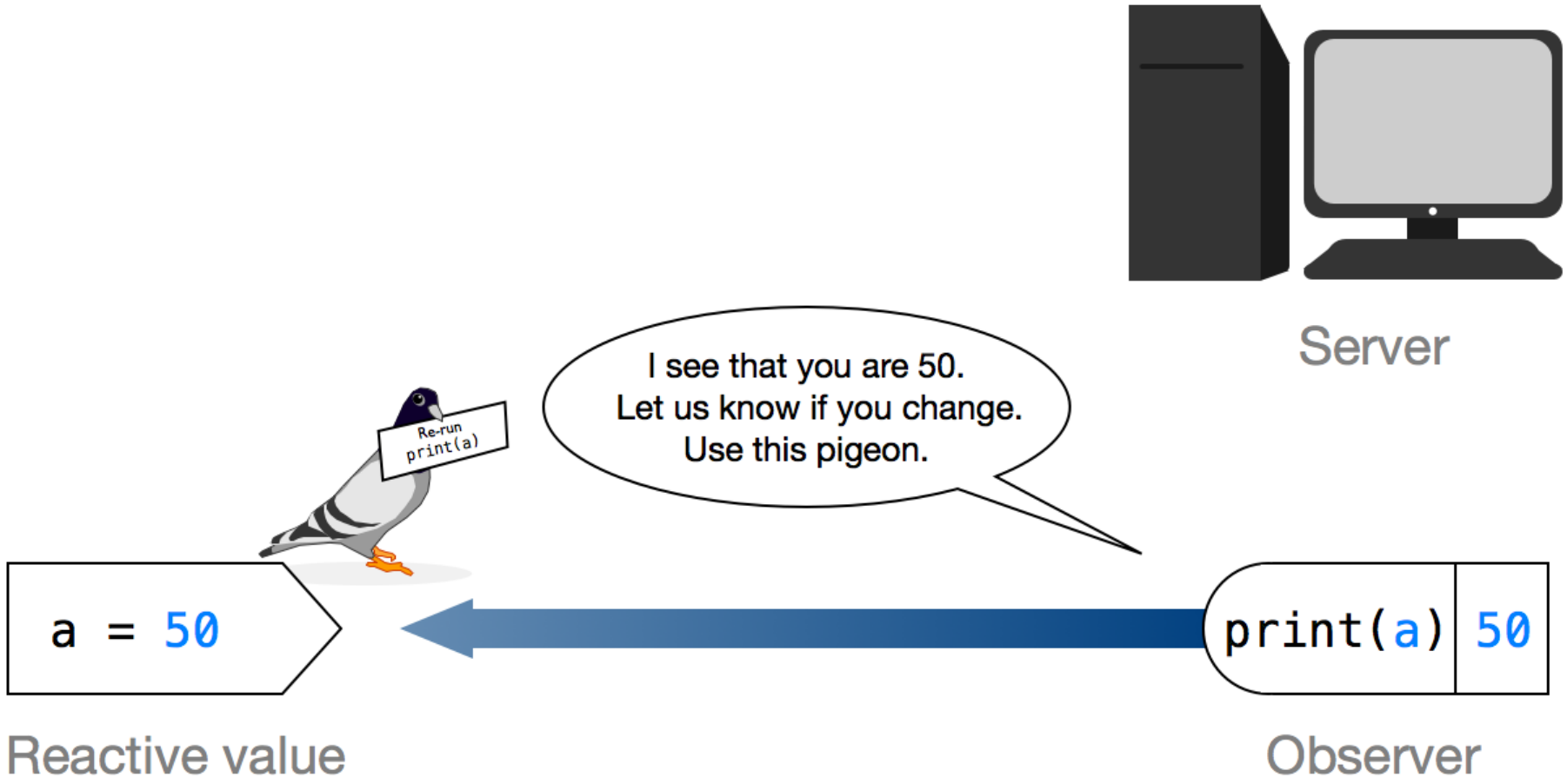
```
server <- function(input, output) {
  # Aquí hacemos el trabajo pesado para que sólo se haga una vez
  pasoIntermedio <- reactive({ trabajo_costoso(input$entrada) })


  # Todos los inputs que usen pasoIntermedio
  output$salidaUno  <- renderText({ pasoIntermedio() })
  output$salidaDos <- renderText({ pasoIntermedio() + 1 })
}
```

Paréntesis al usar el valor (sólo en Shiny)

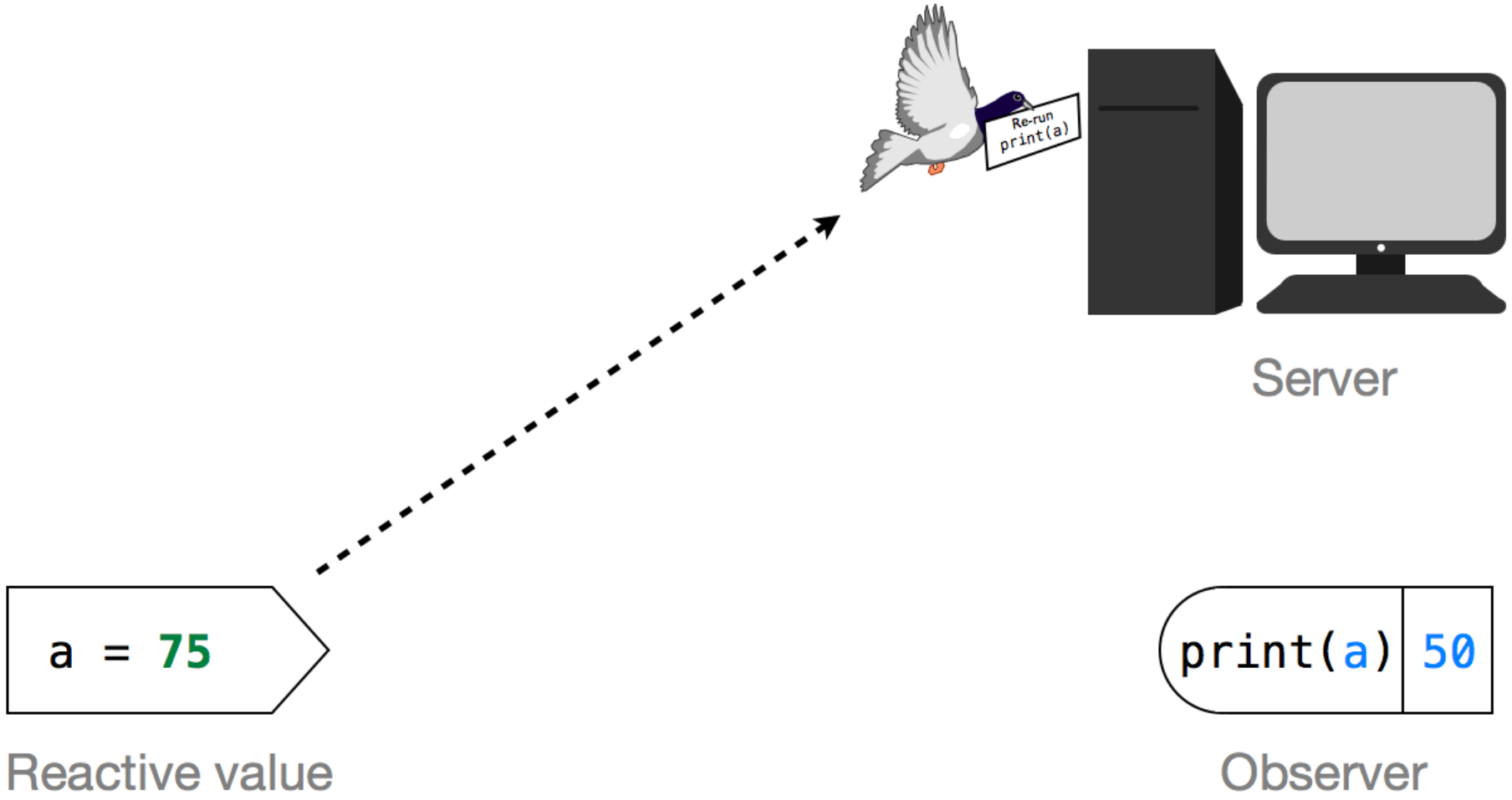
# Resumen de reactividad (Lectura reactividad II)

Link a la lectura

# Cliente / servidor



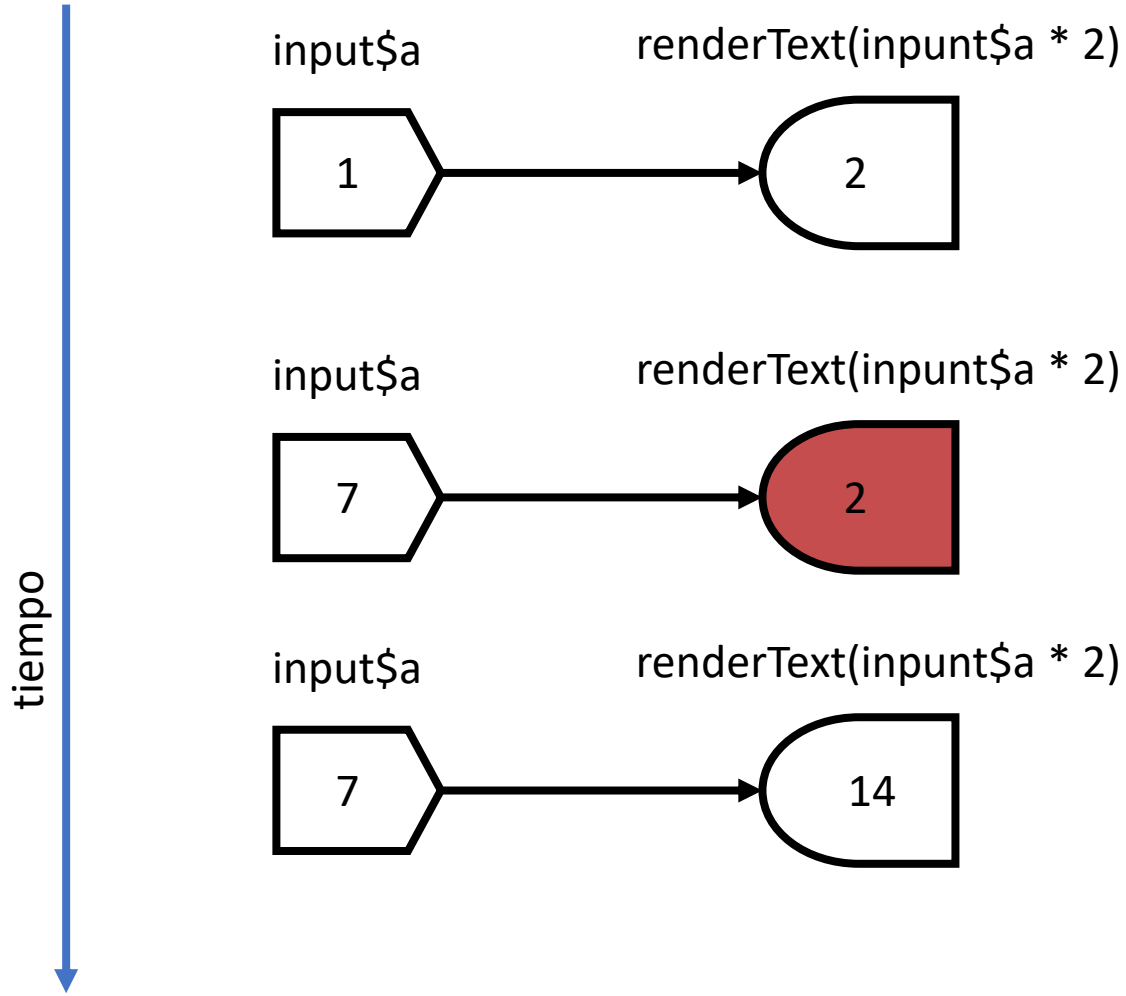
Cuando un cambio ocurre pasa un tiempo. No es instantáneo. Metáfora de la paloma que viaja al servidor)



# Resumen de reactividad (Lectura isolate)

Link a la lectura

# Invalidación



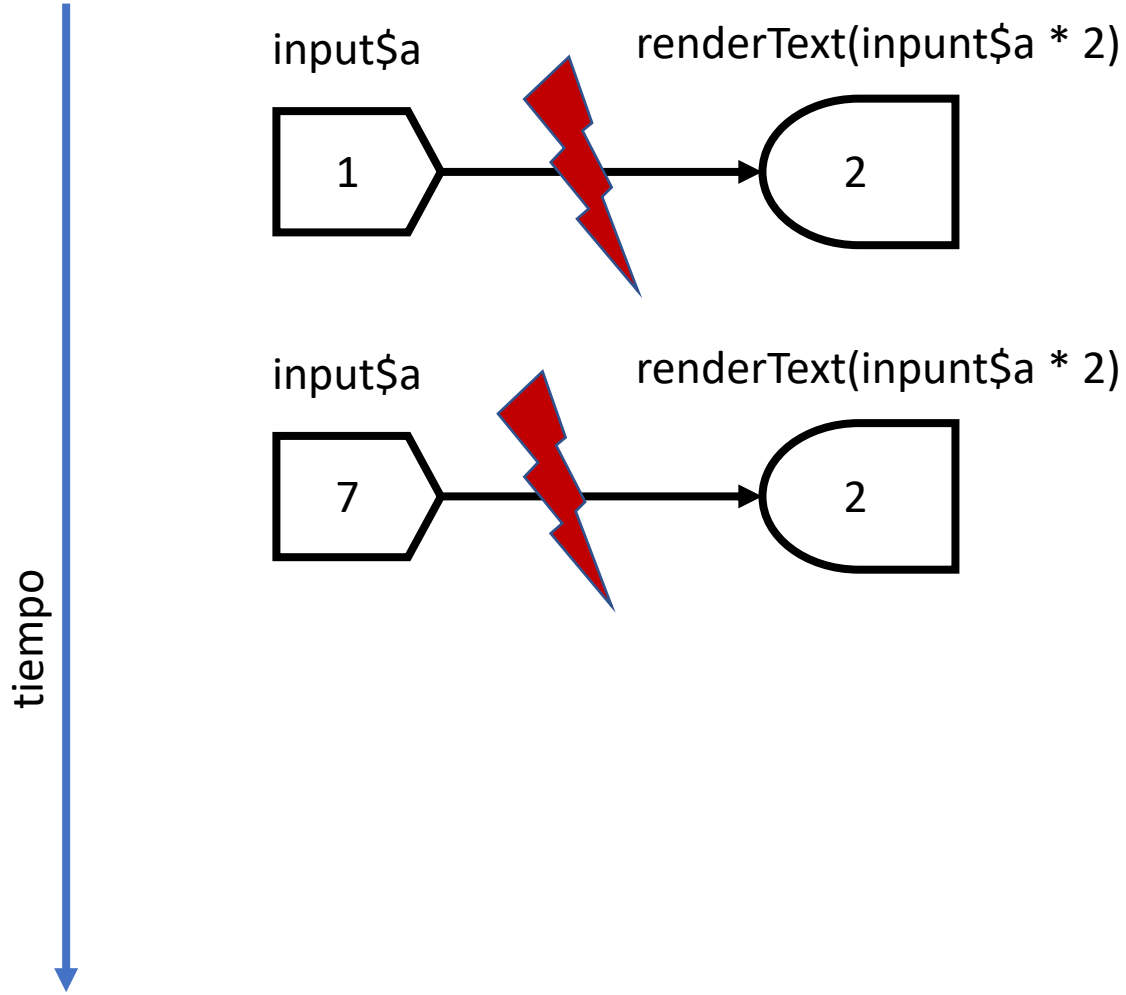

El shiny arranca y se calculan todos los valores reactivos. En este caso el valor por defecto de a es 1 y por tanto el renderText tendrá un 2

El usuario cambia el valor de a (por ejemplo moviendo el slider). TODAS las dependencias de a se invalidan. (La paloma “viaja al servidor”)

El servidor decide qué cosas tiene que recalcular y en qué orden.
En este caso sólo hay que recalcular renderText y devolver el resultado al cliente para que se actualice a 14. El valor de renderText vuelve a estar actualizado

Contenido nuevo

# Isolate: romper la invalidación de esa relación

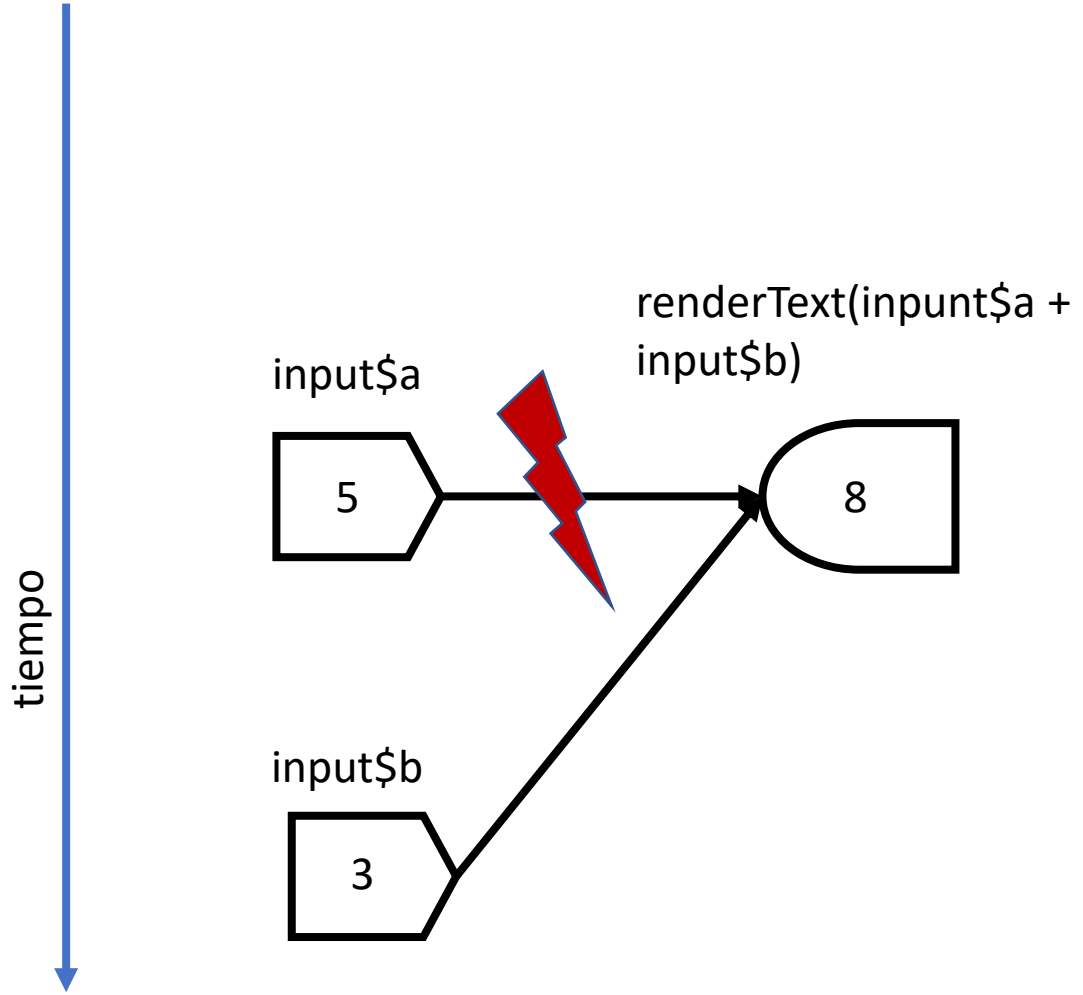



El shiny arranca y se calculan todos los valores reactivos. En este caso el valor por defecto de a es 1 y por tanto el renderText tendrá un 2 **a pesar de que hay un isolate porque es cuando se inicia el programa.**

El usuario cambia el valor de a (por ejemplo moviendo el slider). El render text **NO se actualiza porque isolate rompe esa relación las dependencias**

Contenido nuevo

# Isolate: romper la invalidación de esa relación



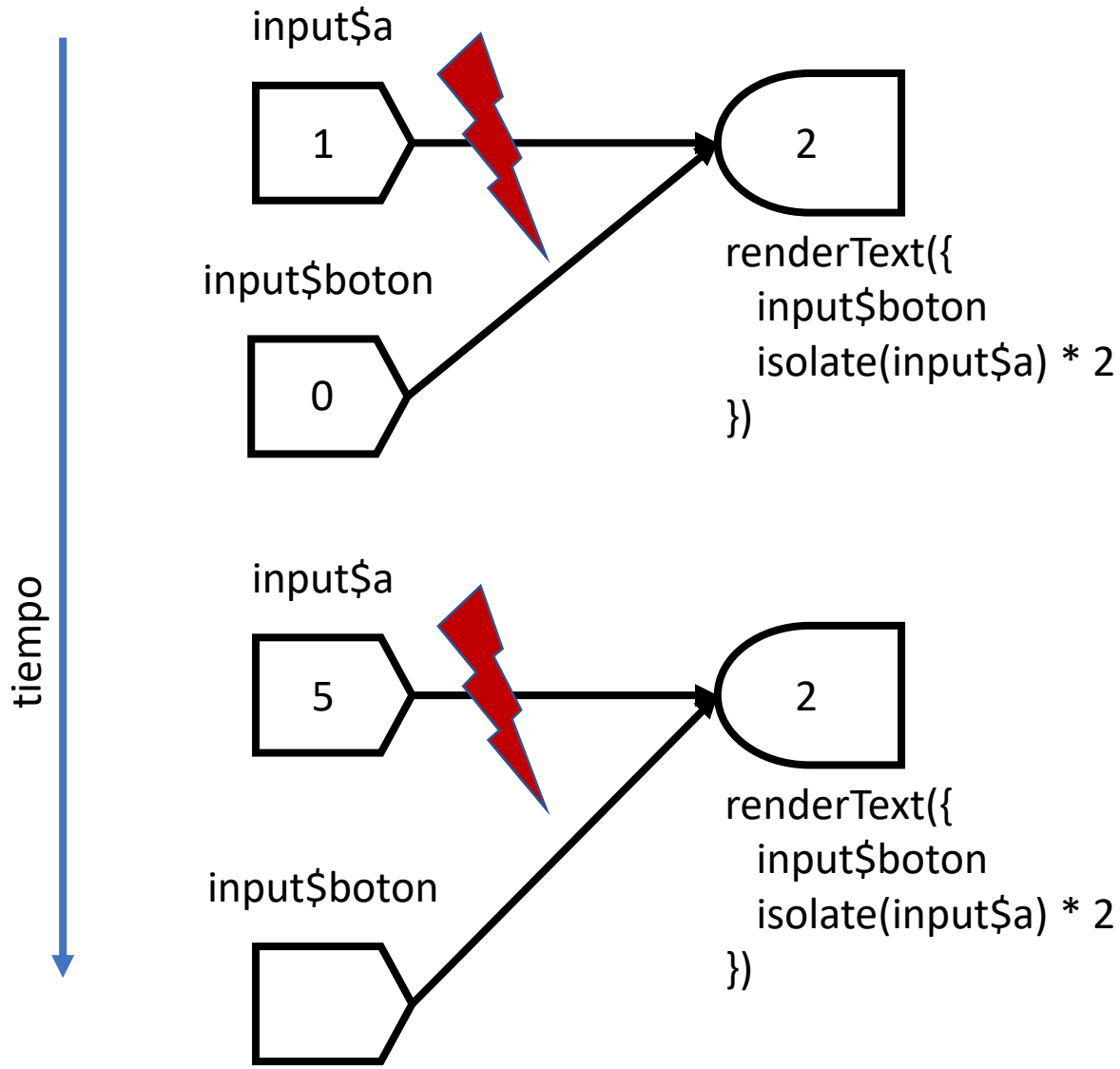

Puedes hacer escenarios más complejos.

¿Qué valor saldría si....?

A) El usuario cambia a = 3
B) El usuario cambia a = 3 y luego b = 3
C) El usuario cambia b = 1 y luego a = 2

Contenido nuevo

# Isolate + actionButton



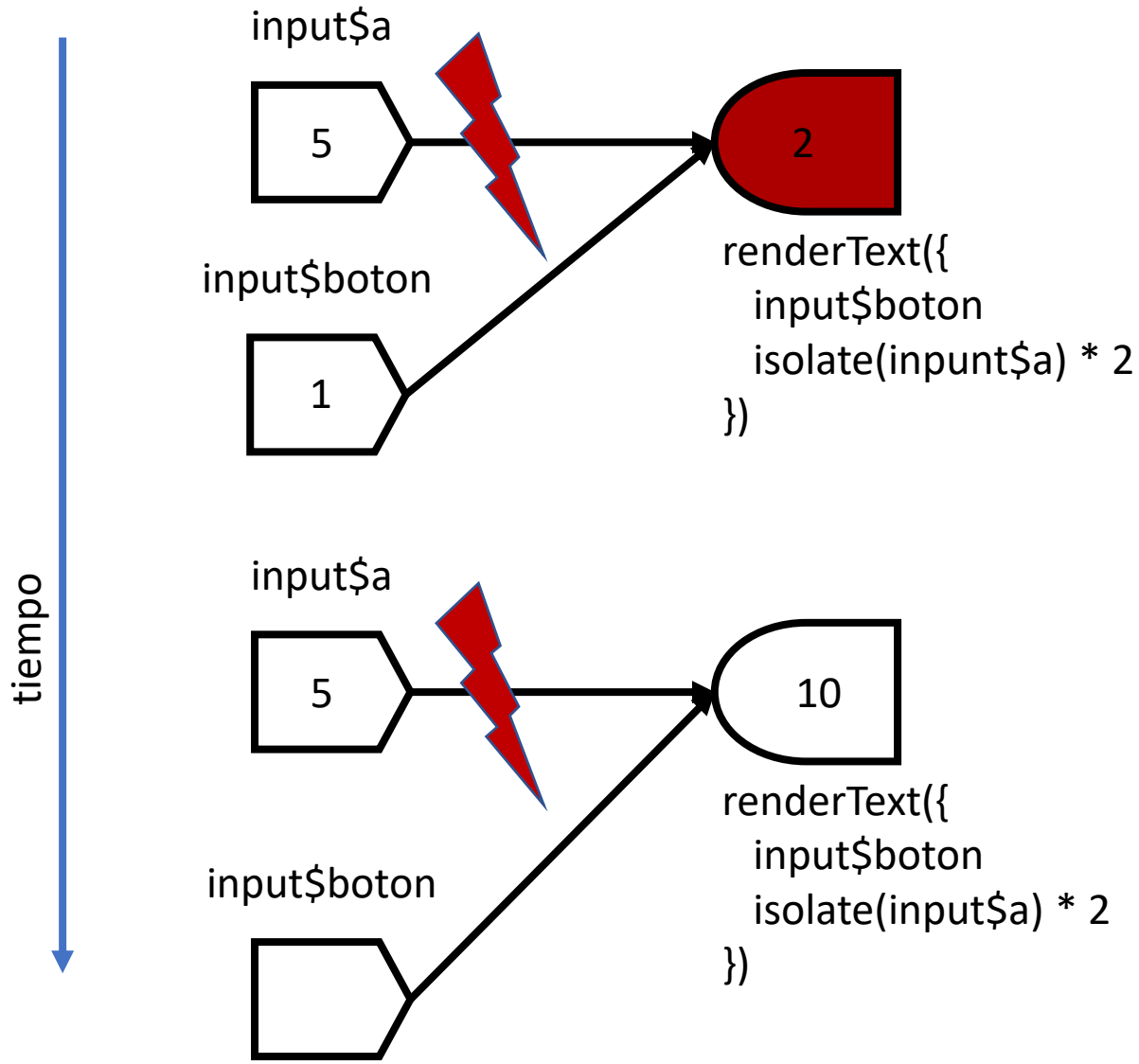

**Como input$botón no se usa en la última línea de las llaves este es ignorado.**

**Los botones tienen como valor el número de veces que ha sido pulsado (ahora mismo 0)**

El usuario cambia el valor de a pero el isolate impide la actualización. No ocurre nada

Contenido nuevo

# Isolate + actionButton




**Al pulsar el botón input$botón incrementa en 1 (ha sido pulsado una vez)**

**Este cambio invalida el renderText**

El renderText se actualiza y repite toda la operación (incluido el input$a * 2)

Contenido nuevo

# Isolate + actionButton

Este escenario es muy común cuando tienes varios inputs que afectan a uno o más outputs.

Por ejemplo: parámetros (que son varios) de un modelo. Si no usaras el isolate + actionButton todo actualizaría por cada cambio, por mínimo que sea.

# Ejercicio 1

Crea un sidebarLayout que tenga en el lateral:

- sliderInput para el tamaño de la muestra
- numericInput para la media
- numericInput para la desviación típica
- Botón para actualizar la gráfica

En el panel central:

- Un histograma de la distribución
- Un summary de la distribucion (en un verbatim text)

Cada vez que el usuario pulsa el botón se genera una nueva muestra aleatoria gaussiana (rnorm)

El histograma y el summary deben calcularse con esa misma muestra.

Sólo se actualiza cuando se pulsa el botón, bajo ninguna otra circunstancia