

# Tolang Playground

## Introduction

Tolang (Tuplespace Operation Language) is a novel programming language designed to orchestrate a batch of concurrently running processes in a distributed environment. This playground packs the Tolang compiler together with the TVM (TupleSpace virtual machine) so you can have a taste of Tolang grammar. To use it to orchestrate a batch of nodes in blockchain, or a group of microservices in your application stack, please contact our sales at [info@concursys.io](mailto:info@concursys.io) for a customized solution to fit your business goal.

A series of distributed computing products built on top of the Tolang executing environment are being developed by ConcurSys Inc. If you are interested to learn more, please contact our sales.

## Running Tolang Playground

- To install Tolang playground, download the binary into Linux and uncompress.
- In the terminal, you can run some Tolang source code. The /examples folder contains some source code you can play with.

```
# Compile and run a Tolang program on command line
$ playground --program "let x in{ x! \"Hello world\" | for (a<-x) { a.log()} }"
"Hello world"

# Compile and run a Tolang source code file
$ playground -f hello_world.to
"Hello world"
```

## Example 1 - a contract factory

This example demonstrates the "reflection" feature of Tolang, a very powerful feature in programming. Here a smart contract factory is defined and stored in the system. Later, any user can call this factory with arguments to obtain a customized version of the smart contract.

- The Tolang source code

```
let contractFactory in {
  //Factory contract, call this to create a simple storage contract
  // input:
  //   retCh: from which to receive a tuple of (contractCode, conInputCh, conAckCh)
  //   contractCode: body of the storage contract
```

```

//      conInputCh: the storage contract's input channel
//      conAck: the storage contract's ack channel
//      ackMsg: used to customize the storage contract's acknowledge message
for ( (retCh, ackMsg) <- contractFactory) {
  let inputCh, ackCh, storeCh in{
    retCh ! {
      (
        for ( (key, value) <- inputCh & ack <- ackCh){
          //"The contract created by the factory is being called".log() |
          storeCh ! (key, value) |
          for ( (=key, a) <- storeCh){
            storeCh ! a |
            ack ! (a, ackMsg)
          }
        },
        inputCh,
        ackCh
      )
    }
  }
} |

let retCh in{
  contractFactory ! (retCh, "Ack message: everything looks good") |
  for( (contractCode, conInputCh, conAckCh) <- retCh ){
    contractCode |
    let ackCh1 in{

      conInputCh ! (1, "Awesome Tolang 1") |
      conAckCh ! ackCh1 |
      for (a <- ackCh1){
        a.log()
      }
    } |
    let ackCh2 in{
      conInputCh ! (2, "Awesome Tolang 2") |
      conAckCh ! ackCh2 |
      for (a <- ackCh2){
        a.log()
      }
    }
  }
}
}

```

- Test it on on playground:

```
$ playground -f playground/examples/contract_factory.to  
( < "Awesome Tolang 1" > < "Ack message: everything looks good" > )  
( < "Awesome Tolang 2" > < "Ack message: everything looks good" > )
```

## Example 2 - the dinning philosophers problem

This example shows the power of Tolang in synchronizing a group of processes competing for shared resources.

In computer science, the dining philosophers problem is an example problem rising from concurrent programming to illustrate the synchronization and deadlock issues. For more details, check:

[https://en.wikipedia.org/wiki/Dining\\_philosophers\\_problem](https://en.wikipedia.org/wiki/Dining_philosophers_problem)

- Tolang source code

Tolang provides a powerful `join` operation primitive to make the solution of this hard problem very easy. Contrary to traditional programming language in which operations (such as resource locking) can only be done sequentially therefore deadlock is inevitable, Tolang's `join` primitive allows developer to model a set of "simultaneous" operations. In the following code, each of the philosopher picks up knife and spoon simultaneous instead of one after another:

```
let placemat1, placemat2, placemat3, placemat4, placemat5 in {  
  // Set the table  
  placemat1 ! "knife1" |  
  placemat3 ! "knife2" |  
  placemat2 ! "spoon1" |  
  placemat4 ! "spoon2" |  
  
  // Philosopher 1's plan  
  for ( "knife1" <- placemat1 & "spoon1" <- placemat2) {  
    "Philosopher 1 is full.".log() |  
    placemat1!("knife1") |  
    placemat2!("spoon1")  
  } |  
  
  // Likewise for philosopher 2  
  for ( "spoon1" <- placemat2 & "knife2" <- placemat3) {  
    "Philosopher 2 is full.".log() |  
    placemat2!("spoon1") |  
    placemat3!("knife2")  
  } |  
  
  // Likewise for philosopher 3  
  for ( "knife2" <- placemat3 & "spoon2" <- placemat4) {
```

```

    "Philosopher 3 is full.".log() |
    placemat3!("knife2") |
    placemat4!("spoon2")
  }|

// Likewise for philosopher 4
for ( "spoon2" <- placemat4 & "knife1" <- placemat1) {
  "Philosopher 4 is full.".log() |
  placemat4!("spoon2") |
  placemat1!("knife1")
}
}

```

- Test it on on playground:

```

$ playground -f examples/dining_philosophers.to
"Philosopher 1 is full."
"Philosopher 3 is full."
"Philosopher 2 is full."
"Philosopher 4 is full."

```

Note that Tolang and TVM guarantees that each philosopher process will and only will execute once. No deadlock or starving can occur. This can be verified by running this code on the playground multiple times, and each time you'll see that all philosophers eat once and only once with different orders!