DistAlgo is a high-level language for writing distributed algorithms, such that they serve as both clear specifications and runnable implementations of those algorithms. This document provides a brief, practical introduction to the major features of the language using several iterations of a familiar example.

DistAlgo is currently implemented as an extension of the Python language, and requires Python 3.4 or higher. This tutorial assumes some familiarity with Object-Oriented programming in general, and with Python, in particular.

1 Main

Let's begin with the simplest possible version of "Hello World" written in DistAlgo.

```
def main():
    print('Hello World.')
```

Listing 1: hello_world01.da - Main Function Definition

Every DistAlgo program must have a main function. In this case, the body of `main` merely contains a call to Python's print function, to output "Hello World" to standard output. As we will see in the next example, the inteded use of the main function in a DistAlgo program is to create, prepare, and then begin the execution of the distinct processes that participate in the distributed algorithm.

2 Processes

In the second version of "Hello World" we can see how process definition and creation works in a DistAlgo program.

```
class P (process):
    def setup(name):
        pass

    def run():
        output('Hello World from:', self.name, ', AKA:', self.id)

def main():
    ps = new(P)
    setup(ps, ('bob',))
    start(ps)
```

Listing 2: hello_world02.da - Process Definition

Distributed algorithms are constituted by the interaction of multiple, distinct processes. DistAlgo is intended for the implementation of distributed algorithms. To that end, DistAlgo makes it easy to define new process types that will execute the behavior required by the algorithm.

Every user defined process type in DistAlgo is an extension of the base class `process`. In the second version of "Hello World" we define a class called `P`, which extends `process`. In order to properly define a new type of DistAlgo process, the user must define two functions: `setup` and `run`.

The `setup` method is used to declare and initialize any instance variables of the user's new process class. Any parameters of the setup method are implicitly declared as instance variables and initialized with the value passed as the argument to the parameter. In the second example, setup has one parameter name, which is initialized with the value 'bob'. In addition, every instance of a distalgo process has a unique identifier stored in an instance variable, self.id, which is the pair of the IP address and the port number upon which the process instance is listening.

The run method of a DistAlgo process contains the definition of the behavior of the process. For the example in listing 2, the process merely prints a “Hello World” message, identifying itself using both the name we supplied as an argument and its unique id.

In this second example we can also see a proper use of the main method of a DistAlgo program. Here we have calls to three Distalgo methods: new, setup, and start.

The new method is used to instantiate a set of DistAlgo process instances. We supply the classname of the process class we wish to instantiate, P, as an argument, and new returns a set of references to the new process instances. In this case we store the set in the variable ps. If the class name is the only argument, then new creates just one instance of that process class. However, even if new is used to create just once process instance, the returned value is still a set–in this case a singleton set–containing a reference to that process instance.

The setup method is used to initialize our new process instances. The first argument is a set of process instance references. The second argument is a tuple of arguments to be passed to the parameters of the setup method defined for the process class corresponding to the class of the process instances one is trying to initialize. Note, even if there is only one argument passed to the setup method of the process class, it must still be packaged within a singleton tuple.

Finally, we can begin the execution of the process instances by passing a set of references to those process instances to the start method. The call to start will cause the run method defined within the corresponding process class to begin executing at each process instance included in the call to start.

In the third example, we can see how multiple process instances are created.

```
class P (process):
    def setup(name):
        pass

    def run():
        output('Hello World from:', name, ', AKA:', self.id)

def main():
    num_processes = 2
    ps = new(P, num = num_processes)
    setup(ps, ('bob', ))
    start(ps)
```

Listing 3: hello_world03.da - Multiple Process Instances

You can run this program from the command line by entering: “python3 -m da hello_world3.da”.

This program is very similar to “hello_world2.da”. In fact, the definition of the process class, P, is identical to the one given in the previous example. There are just two changes in main. First, we add a new variable, num_processes, which is initialized to 2. Second, we pass this new variable as the argument to the num parameter of the new method, resulting in the instantiation of two process instances of the process class, P. These two instances are both initialized by the call to setup, and then set running by the call to start.

The instantiation and execution of the two process instances can be observed in the output when the program is run. There will be two “Hello World” outputs, each with a distinct value for `self.id`, though both will claim that their name is 'bob', since they are both inititialized with the same parameter. You should think about how you might initialize the `name` fields of these two process instances with distinct values.

This latest version of hello_world features two DistAlgo process instances, but they do not interact with or even refer to each other. Distributed algorithms are intended to feature this sort of interplay between distinct processes, so let's add some in the next example.

Here is the fourth example:

```
 1 class P (process):
 2     def setup(processes):
 3         pass
 4
 5     def run():
 6         for p in processes:
 7             output(self.id, 'says Hello to: ', p)
 8
 9 def main():
10     num_processes = 2
11     ps = new(P, num = num_processes)
12     setup(ps, (ps,))
13     start(ps)
```

Listing 4: hello_world04.da - Multiple Processes Referring to Each Other

You can run this program from the command line by entering: "python3 -m da hello_world04.da". There are several changes to the program. First, we have altered the definition of the `setup` method so that it accepts parameter, `processes`, which will hold a set of references to DistAlgo process instances. Next, we have altered the the `run` method of `P` so that instead of greeting the entire world, it instead loops over all the process instances in `processes` and sends each of them an individualized greeting, ¡process id¿ says Hello to ¡process id¿.

Within the main method we generate two distinct process instances of type `P`, and store the set of their references in textttps. We then initialize both instances passing that same set of references to `setup` to serve as the argument to textttprocesses. Finally, we start those process instances running as before.

If you run the program then you will see that in the ouput there are four "X say Hello to Y". If we label the two process instances, $p1$ and $p2$, then the four greetings have the following forms: i) "p1 says Hello to p1"; ii) "p1 says Hello to p2"; iii) "p2 says Hello to p1"; iv) "p2 says Hello to p2", in some arbitrary order. But, we probably did not intend to have the processes greeting themselves. So what went wrong?

The problem occurs on line 12. We passed the entire set ps to the `processes` parameter of `P` for each process instance, but there is a reference to that process instance in the set `ps`. So, each ends up with a `processes` set that contains a reference to itself. How can we avoid this?

One way to address this problem is to prevent a process instance from contacting itself while it executes its `run` method. We can do this with a conditional, as illustrated in Example 5:

```
  class P (process):
      def setup(processes):
          pass

      def run():
          for p in processes:
                if p != self.id:
                    output(self.id, 'says Hello to: ', p)

    def main():
        num_processes = 2
        ps = new(P, num = num_processes)
        setup(ps, (ps,))
        start(ps)
```

Listing 5: hello_world05.da - Excluding Self-References, the Ugly Way

The only change made from Example 4 is the insertion of an `if` statement within the loop of the `run` method that sends the greetings to the processes in the set `processes`. This `if` only allows the process instance to execute the `output` statement if the process id contained in the loop control variable, `p`, is distinct from the process id of the process instance executing the code (`self.id`).

Adding this conditional statement solves our problem, but it does so in a crude and inefficient manner. Real distriubted algorithms will involve more complicated and more numerous interactions between process instances. In order to ensure that a process never attempted to unecessarily interact with itself each interaction statement in the program would have to be guarded by a conditional like this one. That is many extra lines of code that need to be carefully checked to guarantee you have not damaged the control logic of the program. We need a better solution than this.

It turns out there is a common pattern present in many DistAlgo examples that allows one to initialize a process with a set of references to other process instances, but exclude a reference to the process instance one is attempting to initialize. It is illustrated in Example 6:

```
class P (process):
    def setup(processes):
        pass

    def run():
        for p in processes:
        output(self.id, 'says Hello to: ', p)

def main():
    num_processes = 2
    ps = new(P, num = num_processes)
    for p in ps:
        setup(p, (ps - {p}, ))
    start(ps)
```

Listing 6: hello_world06.da - Excluding Self-References, the Nice Way

The only difference between Example 6 and Example 4 occurs inside the call to the `setup` method within main. We initialize each of the process instances one at a time within a loop. For each process instance, instead of passing the entire set `ps` to the `processes` paramter of `P`, we instead pass the set produced taking the difference of `ps` with a singlton set that contains a reference to the process instance being initialized. Thus, each process instance is set up with a set of refereces to every other process instance in `ps`.

If you run Example 6 by entering “python3 -m da hello_world06.da”, then you will see that there now only two outputs. $p1$ greets $p2$, and $p2$ greets $p1$. All is once again right with the world. Each process instance greets the other without also greeting itself.

There are many examples of DistAlgo programs in which set difference is used in this way to create a set of processes that includes all the processes in a set to which the current process belongs, but which excludes the current process. This is not surprising. Process classes define instances that will interact with each other, but they do not usually need to interact with themselves. It would be ugly to check the identity of the process the current instance is attempting to interact with to verify that it is not unnecessarily communicating with itself. Using set difference in this way lets us avoid all that ugliness.

3 Process Interaction

So far we have seen how to define the process classes that will play the roles in the distributed algorithm we are attempting to specify. However, distributed algorithms require these roles to interact by sending and receiving messages among each other. In this section we will introduce the DistAlgo constructs that enable process instances to send messages to and receive messages from each other.

Here is Example 7:

```
class P (process):
    def setup(processes):
        pass

    def run():
        for p in processes:
            send(('hello', self.id), to = p)

def main():
    num_processes = 2
    ps = new(P, num = num_processes)
    for p in ps:
        setup(p, (ps - {p}, ))
    start(ps)
```

Listing 7: hello_world07.da - Send

This program, "hello_world07.da" is basically identical to the program given in Example 6. The only difference is the statement inside the `for` loop inside the `run` method of process `P`, where we have replaced the `output` statement with a new kind of statement `send`.

The `send` method is available to every class that subclasses the DistAlgo `process` class. The `send` method take two arguments. The first argument is the content of the message to be sent. This is typically a tuple of various values. By convention, the first value in the tuple is a string used as a tag for the message–it identifies the type of message being sent. In this example, we have tagged the message in the `send` with the value `'hello'`. This will let the recipient know that it is a greeting message.

The second argument, prefaced by "`to=`", is the process id of the process instance that is the intended recipient of the message. The sending process instance will open a socket to the stated recipient process using its id (recall that a process id is a tuple of the ip address and port number upon which that process is listening for incoming connections).

If we run this program with the command "python3 -m da hello_world07.da", then you will see that there is no observable output. What has happened? The problem is that sending messages is only half of the work we need to do in order to make these process instances genuinely interact. We also need to specify what a process instance should do when it *receives* a message with a `'hello'` tag.

This leads us to Example 8:

```
class P (process):
    def setup(processes):
        pass

    def run():
        for p in processes:
            send(('hello', self.id), to = p)

    def receive(msg=('hello', pid), from_ = greeter):
        output(greeter, 'says Hello to', self.id)

def main():
    num_processes = 2
    ps = new(P, num = num_processes)
    for p in ps:
        setup(p, (ps - {p}, ))
    start(ps)
```

Listing 8: hello_world08.da - Receive Handler

The program in Example 8 is derived from the program in Example 7 by adding a single new construct. This is the receive handler. In the Python implementation of DistAlgo a receive handler is the definition of a method `def receive()`. The parameters of the `receive` method determine the content of the message it is intended to receive and the identity of the sender. In particular, the first parameter–the `msg=` parameter– defines a pattern that a message must match in order to activate the body of the receive handler. The body of the `receive` method determines the behavior of the process instance upon receiving such a message.

In Example 8, the receive handler pattern includes the literal string tag 'hello' and an additional parameter, `pid`. Thus, this receive handler will be activated by the receipt of any message whose first value is the string 'hello', and which has only one other value. That second value will be assigned to the variable `pid` so that it can be used within the body of the receive handler. (Note: it is expected that the argument to the `pid` parameter will be a DistAlgo process instance id, but this will not be enforced by Python's dynamic typing system. If the type of the parameter matters in your program you will have to enforce it manually in the receive handler body.)

Observe that this exactly matches the structure of the message that is being sent in the `run` method above. This is intentional. In order for sent messages to be handled correctly, the pattern defined for the `msg=` parameter must be able to match the structure and tag of the content of the sent message. If the message we sent had contained a different tag, or a different number of additional values, then there would be no match and the body of this receive handler would not execute.

So, let's run this version of the hello_world program by entering: "python3 -m da hello_world08.da", and observe the results. Except, once again nothing appears to happen. What has gone wrong? Nothing has gone wrong, but we need to consider more carefully how message handling in a language built for distributed algorithms should work.

It cannot be the case that messages can be handled at any point during program execution.

Consider the possibility that the execution of the body of the `run` method (or, code at other locations in the program) may depend upon values that can be altered by the receipt of incoming messages. If multiple incoming messages are handled willy-nilly, then these values can be changed before the computations that depend upon them can be fully evaluated. Message handling may interfere and ruin the execution of our program if it is not constrained in some way.

The solution in DistAlgo is to introduce particular locations within the code at which incoming messages (if any) may be handled. These locations are called yield points and they are declared within the program using labels. In order to understand what you are doing when you include a label recognize that the code between two labels is treated as atomic–that code will complete execution without any interruption by messaging events included in the distributed algorithm. It is important to keep this faux atomicity in mind when deciding where to place labels in your program.

In Example 9 we add a label to our hello_world program to handle ensure the greeting message gets handled. Here it is:

```
class P (process):
    def setup(processes):
        pass

    def run():
        for p in processes:
            send(('hello', self.id), to = p)
        -- rcv_greeting

    def receive(msg=('hello', pid), from_ = greeter):
        output(pid, 'says Hello to', self.id)

def main():
    num_processes = 2
    ps = new(P, num = num_processes)
    for p in ps:
        setup(p, (ps - {p}, ))
    start(ps)
```

Listing 9: hello_world09.da - Yield Points

Try running this program several times. You should notice that something is amiss. Most of the time you will see that the process instances exchange greeting messages. However, on some runs of the program, only one of the greeting messages gets output. This is because only one of the greeting messages is handled by its recipient. Why does this happen?

Network communication can take an arbitrary amount of time to complete. You cannot count on messages arriving in a timely fashion in your algorithm (at least, not without doing a lot more work). In the runs featuring the missing output, the process instance that fails to handle the message reaches the label `--rcv_greeting` before the greeting message actually arrives. Since there are no messages waiting when it reaches the label, there is nothing to handle, and so the receive handler is not invoked. The execution falls through past the label, and since there is no further code in the `run` method the process instance completes execution and ends.

In order to gain more control over process interaction we need to do more than merely create opportunities for one process instance to handle messages that may have arrived from the other. We need to synchronize the behavior of the two processes to create genuine interaction in this case. We will accomplish this by using the `await` keyword. Declaring an await statment will allow us to halt the progress of the `run` function while we wait for some specified condition to obtain.

We can see how this is applied to the present problem by looking at Example 10:

```
class P (process):
    def setup(processes):
        pass

    def run():
        for p in processes:
            send(('hello', self.id), to = p)
        -- rcv_greeting
        await(False)

def receive(msg=('hello', self.id), from_ = greeter):
    output(greeter, 'says Hello to', self.id)

def main():
    num_processes = 2
    ps = new(P, num = num_processes)
    for p in ps:
        setup(p, (ps - {p}, ))
    start(ps)
```

Listing 10: hello_world10.da - Synchronizing with Await Statements