CSE 535 Asynchronous Systems
Design Document

Improving and Evaluating Pastry Implementations In DistAlgo

Team 30:
Akhil Bhutani
Nikhil Navadiya
Vivek Kumar Sah

Supervised by:
Prof. Annie Liu

Overall Architecture

Class diagram

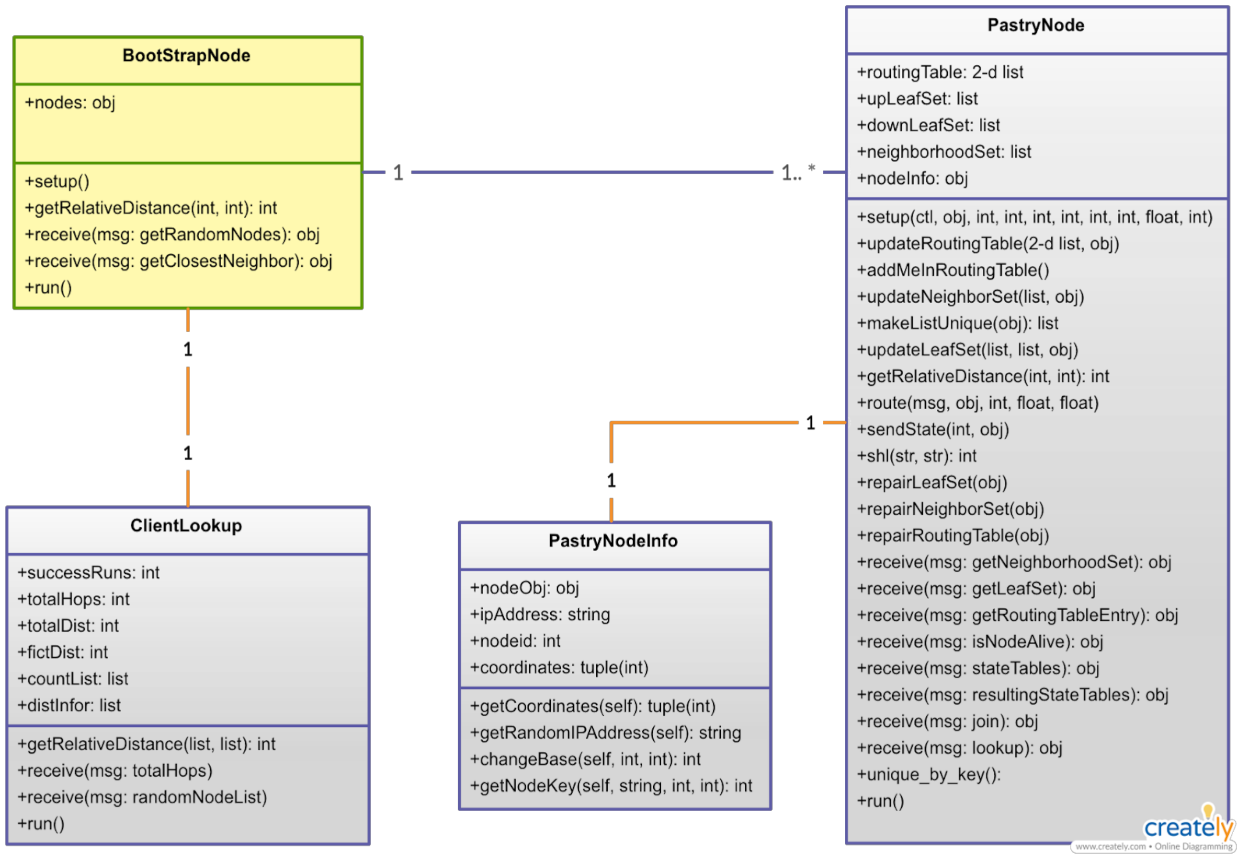
BootStrapNode
+nodes: obj
+setup()
+getRelativeDistance(int, int): int
+receive(msg: getRandomNodes): obj
+receive(msg: getClosestNeighbor): obj
+run()
1
1.. *
1
1
PastryNode
+routingTable: 2-d list
+upLeafSet: list
+downLeafSet: list
+neighborhoodSet: list
+nodeInfo: obj
+setup(ctl, obj, int, int, int, int, int, int, float, int)
+updateRoutingTable(2-d list, obj)
+addMeInRoutingTable()
+updateNeighborSet(list, obj)
+makeListUnique(obj): list
+updateLeafSet(list, list, obj)
+getRelativeDistance(int, int): int
+route(msg, obj, int, float, float)
+sendState(int, obj)
+shl(str, str): int
+repairLeafSet(obj)
+repairNeighborSet(obj)
+repairRoutingTable(obj)
+receive(msg: getNeighborhoodSet): obj
+receive(msg: getLeafSet): obj
+receive(msg: getRoutingTableEntry): obj
+receive(msg: isNodeAlive): obj
+receive(msg: stateTables): obj
+receive(msg: resultingStateTables): obj
+receive(msg: join): obj
+receive(msg: lookup): obj
+unique_by_key():
+run()
1
1
ClientLookup
+successRuns: int
+totalHops: int
+totalDist: int
+fictDist: int
+countList: list
+distInfor: list
+getRelativeDistance(list, list): int
+receive(msg: totalHops)
+receive(msg: randomNodeList)
+run()
PastryNodeInfo
+nodeObj: obj
+ipAddress: string
+nodeid: int
+coordinates: tuple(int)
+getCoordinates(self): tuple(int)
+getRandomIPAddress(self): string
+changeBase(self, int, int): int
+getNodeKey(self, string, int, int): int
creately
www.creately.com • Online Diagramming

Fig. 1 Class Diagram Representing all Classes

The class diagram presented in figure 1 represents the classes defined in our program. This also depicts the relationship with various classes and member functions of each classes along with global variables accessible to each method.

Sequence diagram for Client Lookup

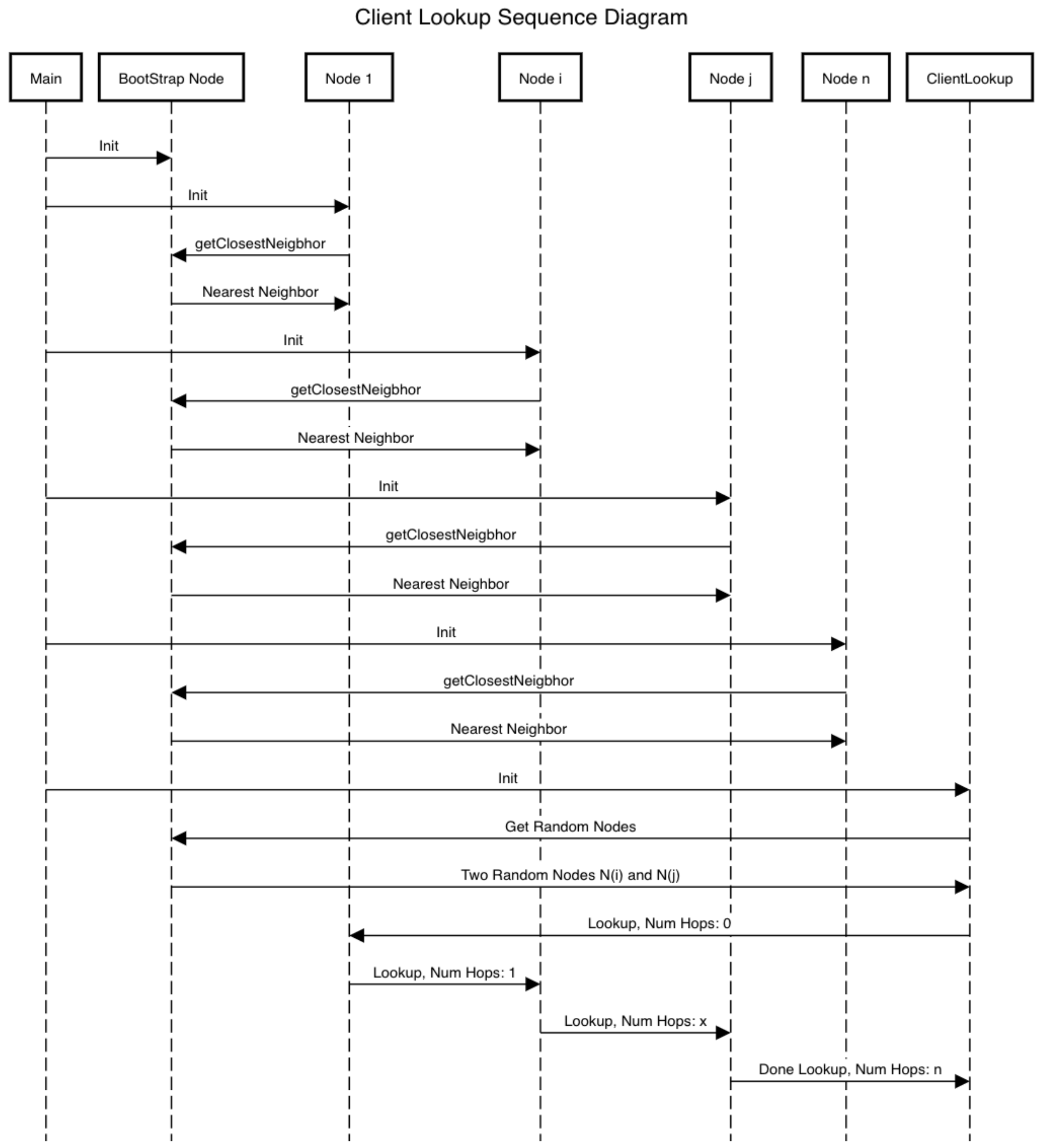

Fig. 2 Client Lookup Sequence Diagram

FlowChart

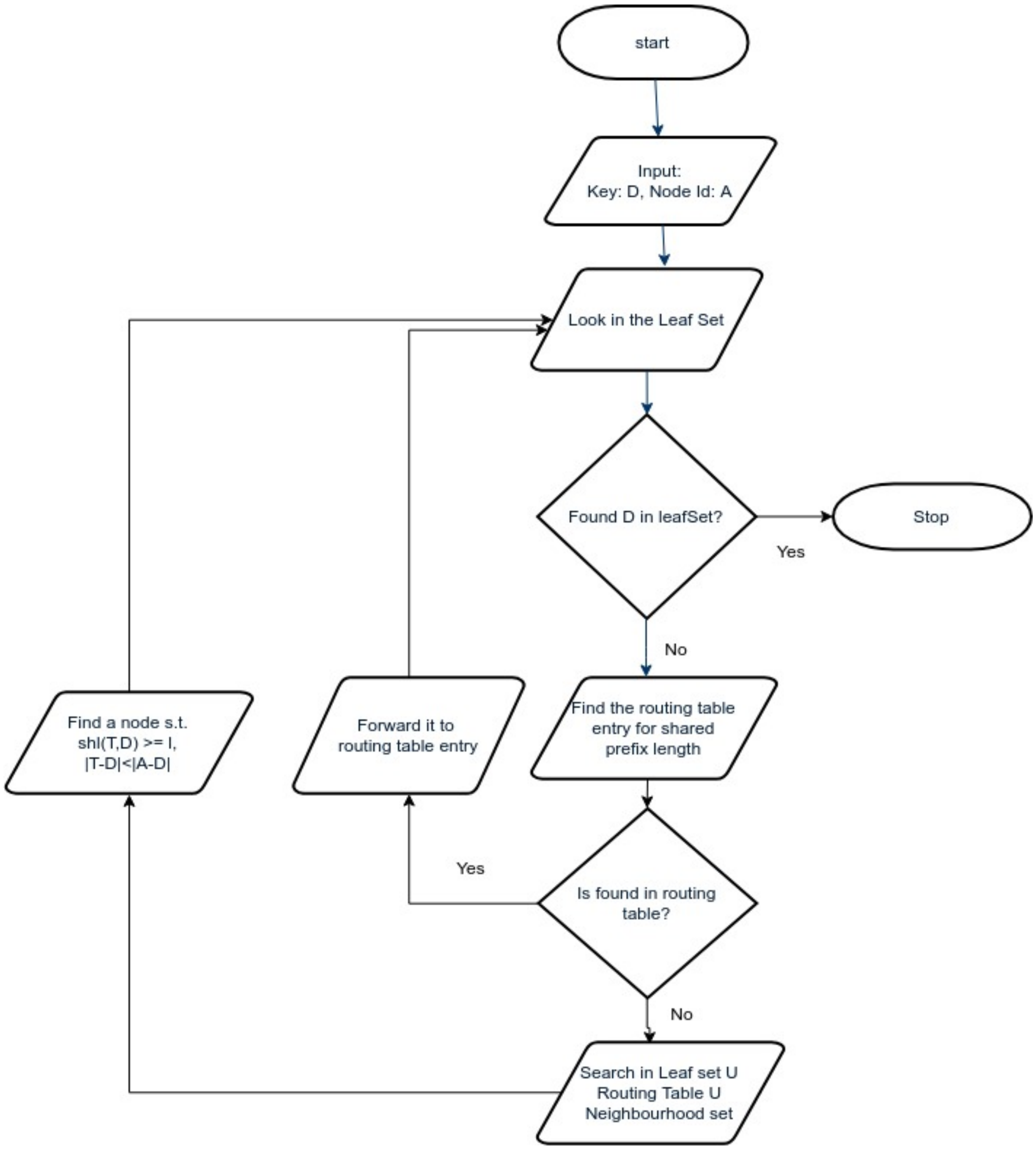
start
Input:
Key: D, Node Id: A
Look in the Leaf Set
Found D in leafSet?
Yes
Stop
No
Find the routing table entry for shared prefix length
Is found in routing table?
Yes
Forward it to routing table entry
No
Search in Leaf set U Routing Table U Neighbourhood set
Find a node s.t. shl(T,D) >= l, |T-D|<|A-D|

Fig. 3 Flow Chart representing Pastry Routing Algorithm

Design Summary

This section presents a brief summary for the overall design followed in our implementation. It discusses the driver program, important APIs that are essential to the Pastry core routing algorithm.

Driver Program:

The driver program reads the necessary arguments from the command line and initializes various parameters as part of the Pastry algorithm. The various parameters have been described in section 3.4. It initializes a bootstrap node which will provide assistance to any new nodes joining the Pastry network. It will provide the new node with a nearby node whom it can contact for sending the join request. In addition to the above, it will provide random nodes to clients querying for lookup as it maintains all the nodes information in the pastry network. This is done in DistAlgo as:

```
# Initialize the bootstrap node
boot = new(pastry.BootStrapNode)
setup(boot, (b, keySize, L, M, leafSetLen, tack))
start(boot)
```

The next step is to initialize the nodes in the Pastry network. The number of nodes is provided as an argument by the user. Various timing parameters are initialized during this phase. The code for the same in DistAlgo is:

```
# Initialize Pastry nodes
startTimeNodeInit = time.time()
ps = new(pastry.PastryNode, num=N)
totalTimeJoin = 0.0
```

Next, we initialize a controller to measure total user time, total system time, total process time, total memory and total wall clock time.
Using the below piece of code, we send wait for ‘JoinCompleted’ from all pastry nodes:

```
count = 0
for p in ps:
    endTimeNodeJoin = time.time()
    startTimeNodeJoin = time.time()
    setup(p, (ctl, boot, b, keySize, L, M, leafSetLen, ti, tack, tn))
    start(p)
    if await(some(received(('JoinCompleted'), from_= p))):
        count += 1
        endTimeNodeJoin = time.time()
        pass
    elif timeout(5):
        endTimeNodeJoin = time.time()
        pass
    time.sleep(0.1)
    totalTimeJoin += (endTimeNodeJoin - startTimeNodeJoin)
```

```
 # Only count nodes are initialized. Any node with
same          nodeid won't affect
# count variable. Hence, N = count
N = count
endTimeNodeInit = time.time()
```

Now, N is the number of nodes which successfully joined the network.

Next, based on the user supplied parameter, we randomly kill dNodes and start lookup between any two nodes. We use the end keyword in DistAlgo to kill a process and get random nodes from bootstrap node.

For ClientLookup class, we get two random nodes from bootstrap node. We always make sure that bootstrap node returns us alive source and destination nodes based on inherent logic present in pastry.da line number 33 - 51. Then a request is sent to the source node and routing continues as per the algorithm presented in the paper. [1] The below code represents the snippet for initializing the ClientLookup class which performs lookup among two random nodes.

```
# Start Lookup Process
cl = new(ClientLookup, num=1)
setup(cl, (boot, N, lTrials, tl))
start(cl)
```

Some of the important APIs that are implemented as part of Pastry core routing algorithm are -

1. *def route(message, dstObj, numHops, tt, tDist)* : Routes a message towards destination node ‘dstObj’. The number of hops to reach the current node is recorded in numHops. The distance traveled so far to reach the current node is recorded in tDist. tt is the lamport’s logical clock of the source node to distinguish lookups. This is used as there can be conflict in the received variable provided in DistAlgo and same message may already be present in which case it won’t initiate the lookup.

2. *def sendState(numHops, X):* This method is used to send the resulting state to a node X once a node joins the network. The number of hops is stored in numHops.

3. *def shl(A, B):* This is a helper function which calculates the shared prefix length between two nodes A and B.

4. *def makeListUnique(objSet):* This method acts as a helper function to filter out a list of objects and make them unique based on node Ids. It takes a list as an input and returns a list with unique node information in output. This is needed as we are passing a copy of the class PastryNodeInfo which contains basic information of each node. This copy cannot be filtered based on memory location as it may be different when it is passed to another node.

5. *update* and repair* functions:* These functions perform the act of updating / repairing the state tables of a node when they are corrupted or a new node information is provided to them.

The link to documentation generated using PyDoc is presented below:

https://github.com/unicomputing/pastry-distalgo/tree/master/PyDoc

The above link contains two files main.html and pastry.html which is the documentation generated for main.da and pastry.da respectively.

main

index
/home/bhutani92/Desktop/AsynchronousSystems/pastry-distalgo/main.py

```
# -*- generated by 1.0.12 -*-
```

Modules

da	os	random	time
numpy	pastry	sys	

Classes

da.sim.DistProcess(builtins.object)

    ClientLookup

da.sim.NodeProcess(da.sim.DistProcess)

    Node_

class **ClientLookup**(da.sim.DistProcess)

```
Abstract base class for DistAlgo processes.

Each instance of this class enbodies the runtime state and activities of a
DistAlgo process in a distributed system. Each process is uniquely
identified by a `ProcessId` object. Messages exchanged between DistAlgo
processes can be any picklable Python object.

DistAlgo processes can spawn more processes by calling `new`. The process
that called `new` is known as the parent process of the newly spawned
processes. Any DistProcess can send messages to any other DistProcess, given
that it knows the `ProcessId` of the target process. However, only the
parent process can `end` a child process. The terminal is shared between all
processes spawned from that terminal, which includes the stdout, stdin, and
stderr streams.

Concrete subclasses of `DistProcess` must define the methods:

- `setup`: A function that initializes the process-local variables.

- `run`: The entry point of the process. This function defines the
  activities of the process.

Users should not instantiate this class directly, process instances should
be created by calling `new`.
```

Method resolution order:
- ClientLookup
- da.sim.DistProcess
- builtins.object

Methods defined here:

__init__(self, procimpl, props)
```
Initialize self.  See help(type(self)) for accurate signature.
```

getRelativeDistance(self, coord1, coord2)

run(self)
```
Entry point for the DistAlgo process.

This is the starting point of execution for user code.
```

setup(self, boot, N, lTrials, tl, tps, **rest_1112)
```
Initialization routine for the DistAlgo process.

Should be overridden by child classes to initialize process states.
```

Methods inherited from da.sim.DistProcess:

__repr__(self)
```
Return repr(self).
```

__str__ = __repr__(self)
```
Return repr(self).
```

debug(self, *message, sep=' ')
```
Prints debugging output to the process log.

This is the same as `output` except the message is logged at the
'USRDBG' level.
```

end(self, target, exit_code=1)
```
Terminate the child processes specified by `target`.

`target` can be a process id or a set of process ids, all of which must
be a child process of this process.
```

error(self, *message, sep=' ')
```
Prints error message to the process log.

This is the same as `output` except the message is logged at the
'USRERR' level.
```

exit(self, code=0)
```
Terminates the current process.

`code` specifies the exit code.
```

hanged(self)
```
Hangs the current process.

When a process enters the 'hanged' state, its main logic and all message
handlers will no longer run.
```

incr_logical_clock(self)
```
Increments the logical clock.

For Lamport's clock, this increases the clock value by 1.
```

logical_clock(self)
```
Returns the current value of the logical clock.
```

nameof(self, pid)
```
Returns the process name of `pid`, if any.
```

new(self, pcls, args=None, num=None, at=None, method=None, **props)
```
Creates new DistAlgo processes.

`pcls` specifies the DistAlgo process class. Optional argument `args` is
a list of arguments that is used to call the `setup` method of the child
processes. Optional argument `num` specifies the number of processes to
create on each node. Optional argument `at` specifies the node or nodes
on which the new processes are to be created. If `num` is not specified
then it defaults to one process. If `at` is not specified then it
defaults to the same node as the current process. Optional argument
`method` specifies the type of implementation used to run the new
process(es), and can be one of 'process', in which case the new
processes will be run inside operating system processes, or 'thread' in
which case the processes will be run inside operating system threads. If
method is not specified then its default value is taken from the
'--default_proc_impl' command line option.

If neither `num` nor `at` is specified, then `new` will return the
process id of child process if successful, or None otherwise. If either
`num` or `at` is specified, then `new` will return a set containing the
process ids of the processes that was successfully created.
```

nodeof(self, pid)
```
Returns the id of the node process that's running `pid`.
```

output(self, *message, sep=' ', level=21)
```
Prints arguments to the process log.

Optional argument 'level' is a positive integer that specifies the
logging level of the message, defaults to 'logging.INFO'(20). Refer to
[https://docs.python.org/3/library/logging.html#levels] for a list of
predefined logging levels.

When the level of the message is equal to or higher than the
configured level of a log handler, the message is logged to that
handler; otherwise, it is ignored. DistAlgo processes are
automatically configured with two log handlers:, one logs to the
console, the other to a log file; the handlers' logging levels are
controlled by command line parameters.
```

parent(self)

```
Returns the parent process id.

The parent process is the process that called `new` to create this
process.
```

resolve(self, name)

```
Returns the process id associated with `name`.
```

send(self, message, to, channel=None, **rest)

```
Send a DistAlgo message.

`message` can be any pickle-able Python object. `to` can be a process id
or a set of process ids.
```

work(self)

```
Waste some random amount of time.

This suspends execution of the process for a period of 0-2 seconds.
```

Class methods inherited from da.sim.DistProcess:

get_config(key, default=None) from builtins.type

```
Returns the configuration value for specified 'key'.
```

Data descriptors inherited from da.sim.DistProcess:

__dict__

```
dictionary for instance variables (if defined)
```

__weakref__

```
list of weak references to the object (if defined)
```

Data and other attributes inherited from da.sim.DistProcess:

AckCommands = [<Command.NewAck: 16>, <Command.EndAck: 15>, <Command.StartAck: 11>, <Command.SetupAck: 12>, <Command.ResolveAck: 17>, <Command.RPCReply: 31>]

class **Node_**(da.sim.NodeProcess)

```
Abstract base class for DistAlgo processes.

Each instance of this class enbodies the runtime state and activities of a
DistAlgo process in a distributed system. Each process is uniquely
identified by a `ProcessId` object. Messages exchanged between DistAlgo
processes can be any picklable Python object.

DistAlgo processes can spawn more processes by calling `new`. The process
that called `new` is known as the parent process of the newly spawned
processes. Any DistProcess can send messages to any other DistProcess, given
that it knows the `ProcessId` of the target process. However, only the
parent process can `end` a child process. The terminal is shared between all
processes spawned from that terminal, which includes the stdout, stdin, and
stderr streams.

Concrete subclasses of `DistProcess` must define the methods:

- `setup`: A function that initializes the process-local variables.

- `run`: The entry point of the process. This function defines the
  activities of the process.

Users should not instantiate this class directly, process instances should
be created by calling `new`.
```

Method resolution order:

- Node_
- da.sim.NodeProcess
- da.sim.DistProcess
- builtins.object

Methods defined here:

__init__(self, procimpl, props)

```
Initialize self.  See help(type(self)) for accurate signature.
```

run(self)
```
    Entry point for the DistAlgo process.

    This is the starting point of execution for user code.
```

Methods inherited from da.sim.NodeProcess:

bootstrap(self)

Data and other attributes inherited from da.sim.NodeProcess:

AckCommands = [<Command.NewAck: 16>, <Command.EndAck: 15>, <Command.StartAck: 11>, <Command.SetupAck: 12>, <Command.ResolveAck: 17>, <Command.RPCReply: 31>, <Command.NodeAck: 18>]

Methods inherited from da.sim.DistProcess:

__repr__(self)
```
    Return repr(self).
```

__str__ = __repr__(self)
```
    Return repr(self).
```

debug(self, *message, sep=' ')
```
    Prints debugging output to the process log.

    This is the same as `output` except the message is logged at the
    'USRDBG' level.
```

end(self, target, exit_code=1)
```
    Terminate the child processes specified by `target`.

    `target` can be a process id or a set of process ids, all of which must
    be a child process of this process.
```

error(self, *message, sep=' ')
```
    Prints error message to the process log.

    This is the same as `output` except the message is logged at the
    'USRERR' level.
```

exit(self, code=0)
```
    Terminates the current process.

    `code` specifies the exit code.
```

hanged(self)
```
    Hangs the current process.

    When a process enters the 'hanged' state, its main logic and all message
    handlers will no longer run.
```

incr_logical_clock(self)
```
    Increments the logical clock.

    For Lamport's clock, this increases the clock value by 1.
```

logical_clock(self)
```
    Returns the current value of the logical clock.
```

nameof(self, pid)
```
    Returns the process name of `pid`, if any.
```

new(self, pcls, args=None, num=None, at=None, method=None, **props)
```
    Creates new DistAlgo processes.

    `pcls` specifies the DistAlgo process class. Optional argument `args` is
    a list of arguments that is used to call the `setup` method of the child
    processes. Optional argument `num` specifies the number of processes to
    create on each node. Optional argument `at` specifies the node or nodes
    on which the new processes are to be created. If `num` is not specified
    then it defaults to one process. If `at` is not specified then it
    defaults to the same node as the current process. Optional argument
    `method` specifies the type of implementation used to run the new
    process(es), and can be one of 'process', in which case the new
    processes will be run inside operating system processes, or 'thread' in
    which case the processes will be run inside operating system threads. If
    method is not specified then its default value is taken from the
    '--default_proc_impl' command line option.
```

```
If neither `num` nor `at` is specified, then `new` will return the
process id of child process if successful, or None otherwise. If either
`num` or `at` is specified, then `new` will return a set containing the
process ids of the processes that was successfully created.
```

nodeof(self, pid)

```
Returns the id of the node process that's running `pid`.
```

output(self, *message, sep=' ', level=21)

```
Prints arguments to the process log.

Optional argument 'level' is a positive integer that specifies the
logging level of the message, defaults to 'logging.INFO'(20). Refer to
[https://docs.python.org/3/library/logging.html#levels] for a list of
predefined logging levels.

When the level of the message is equal to or higher than the
configured level of a log handler, the message is logged to that
handler; otherwise, it is ignored. DistAlgo processes are
automatically configured with two log handlers:, one logs to the
console, the other to a log file; the handlers' logging levels are
controlled by command line parameters.
```

parent(self)

```
Returns the parent process id.

The parent process is the process that called `new` to create this
process.
```

resolve(self, name)

```
Returns the process id associated with `name`.
```

send(self, message, to, channel=None, **rest)

```
Send a DistAlgo message.

`message` can be any pickle-able Python object. `to` can be a process id
or a set of process ids.
```

setup(self, **rest)

```
Initialization routine for the DistAlgo process.

Should be overridden by child classes to initialize process states.
```

work(self)

```
Waste some random amount of time.

This suspends execution of the process for a period of 0-2 seconds.
```

Class methods inherited from da.sim.DistProcess:

get_config(key, default=None) from builtins.type

```
Returns the configuration value for specified 'key'.
```

Data descriptors inherited from da.sim.DistProcess:

__dict__

```
dictionary for instance variables (if defined)
```

__weakref__

```
list of weak references to the object (if defined)
```

Functions

sqrt(...)

```
sqrt(x)

Return the square root of x.
```

Data

PatternExpr_1028 = (='LookupCompleted',_)
PatternExpr_1050 = (='LookupCompleted',t)
PatternExpr_273 = (='totalHops',src,dst,numHops,tDist)
PatternExpr_399 = (='randomNodeList',objList)
PatternExpr_406 = X
PatternExpr_457 = (='LookupCompleted',_)
PatternExpr_479 = (='LookupCompleted',t)
PatternExpr_894 = ='JoinCompleted'
PatternExpr_898 = p
PatternExpr_962 = (='randomNodeList',objs)
PatternExpr_969 = X

index
/home/bhutani92/Desktop/AsynchronousSystems/pastry-distalgo/pastry.py

pastry

```
# -*- generated by 1.0.12 -*-
```

Modules

da	math	random	time
hashlib	os	sys	

Classes

builtins.object

    PastryNodeInfo

da.sim.DistProcess(builtins.object)

    BootStrapNode
    PastryNode

class **BootStrapNode**(da.sim.DistProcess)

```
Abstract base class for DistAlgo processes.

Each instance of this class enbodies the runtime state and activities of a
DistAlgo process in a distributed system. Each process is uniquely
identified by a `ProcessId` object. Messages exchanged between DistAlgo
processes can be any picklable Python object.

DistAlgo processes can spawn more processes by calling `new`. The process
that called `new` is known as the parent process of the newly spawned
processes. Any DistProcess can send messages to any other DistProcess, given
that it knows the `ProcessId` of the target process. However, only the
parent process can `end` a child process. The terminal is shared between all
processes spawned from that terminal, which includes the stdout, stdin, and
stderr streams.

Concrete subclasses of `DistProcess` must define the methods:

- `setup`: A function that initializes the process-local variables.

- `run`: The entry point of the process. This function defines the
  activities of the process.

Users should not instantiate this class directly, process instances should
be created by calling `new`.
```

Method resolution order:

    BootStrapNode
    da.sim.DistProcess
    builtins.object

Methods defined here:

__init__(self, procimpl, props)

```
Initialize self.  See help(type(self)) for accurate signature.
```

getRelativeDistance(self, coord1, coord2)

```
For scalar procimity metric, we use geographic distance of pastry nodes
for populating the neighborhood set.

Page number 9, section 2.5, paragraph 2 mentions:
'Pastry’s notion of network proximity is based on a scalar proximity
metric, such as the number of IP routing hops or geographic distance.'
```

run(self)

```
This is the driver program for the BootStrap node. It simply waits for
some process to give the message ExitProcess. Upon receiving this
message, it terminates itself.
```

setup(self, b, keySize, L, M, leafSetLen, tack, **rest_4078)

```
Initialization routine for the DistAlgo process.
```

```
    Should be overridden by child classes to initialize process states.
```

Methods inherited from da.sim.DistProcess:

__repr__(self)
```
    Return repr(self).
```

__str__ = __repr__(self)
```
    Return repr(self).
```

debug(self, *message, sep=' ')
```
    Prints debugging output to the process log.

    This is the same as `output` except the message is logged at the
    'USRDBG' level.
```

end(self, target, exit_code=1)
```
    Terminate the child processes specified by `target`.

    `target` can be a process id or a set of process ids, all of which must
    be a child process of this process.
```

error(self, *message, sep=' ')
```
    Prints error message to the process log.

    This is the same as `output` except the message is logged at the
    'USRERR' level.
```

exit(self, code=0)
```
    Terminates the current process.

    `code` specifies the exit code.
```

hanged(self)
```
    Hangs the current process.

    When a process enters the 'hanged' state, its main logic and all message
    handlers will no longer run.
```

incr_logical_clock(self)
```
    Increments the logical clock.

    For Lamport's clock, this increases the clock value by 1.
```

logical_clock(self)
```
    Returns the current value of the logical clock.
```

nameof(self, pid)
```
    Returns the process name of `pid`, if any.
```

new(self, pcls, args=None, num=None, at=None, method=None, **props)
```
    Creates new DistAlgo processes.

    `pcls` specifies the DistAlgo process class. Optional argument `args` is
    a list of arguments that is used to call the `setup` method of the child
    processes. Optional argument `num` specifies the number of processes to
    create on each node. Optional argument `at` specifies the node or nodes
    on which the new processes are to be created. If `num` is not specified
    then it defaults to one process. If `at` is not specified then it
    defaults to the same node as the current process. Optional argument
    `method` specifies the type of implementation used to run the new
    process(es), and can be one of 'process', in which case the new
    processes will be run inside operating system processes, or 'thread' in
    which case the processes will be run inside operating system threads. If
    method is not specified then its default value is taken from the
    '--default_proc_impl' command line option.

    If neither `num` nor `at` is specified, then `new` will return the
    process id of child process if successful, or None otherwise. If either
    `num` or `at` is specified, then `new` will return a set containing the
    process ids of the processes that was successfully created.
```

nodeof(self, pid)
```
    Returns the id of the node process that's running `pid`.
```

output(self, *message, sep=' ', level=21)
```
    Prints arguments to the process log.

    Optional argument 'level' is a positive integer that specifies the
    logging level of the message, defaults to 'logging.INFO'(20). Refer to
    [https://docs.python.org/3/library/logging.html#levels] for a list of
```

```
predefined logging levels.

When the level of the message is equal to or higher than the
configured level of a log handler, the message is logged to that
handler; otherwise, it is ignored. DistAlgo processes are
automatically configured with two log handlers:, one logs to the
console, the other to a log file; the handlers' logging levels are
controlled by command line parameters.
```

parent(self)

```
Returns the parent process id.

The parent process is the process that called `new` to create this
process.
```

resolve(self, name)

```
Returns the process id associated with `name`.
```

send(self, message, to, channel=None, **rest)

```
Send a DistAlgo message.

`message` can be any pickle-able Python object. `to` can be a process id
or a set of process ids.
```

work(self)

```
Waste some random amount of time.

This suspends execution of the process for a period of 0-2 seconds.
```

Class methods inherited from da.sim.DistProcess:

get_config(key, default=None) from builtins.type

```
Returns the configuration value for specified 'key'.
```

Data descriptors inherited from da.sim.DistProcess:

__dict__

```
dictionary for instance variables (if defined)
```

__weakref__

```
list of weak references to the object (if defined)
```

Data and other attributes inherited from da.sim.DistProcess:

AckCommands = [<Command.NewAck: 16>, <Command.EndAck: 15>, <Command.StartAck: 11>, <Command.SetupAck: 12>, <Command.ResolveAck: 17>, <Command.RPCReply: 31>]

class **PastryNode**(da.sim.DistProcess)

```
Abstract base class for DistAlgo processes.

Each instance of this class enbodies the runtime state and activities of a
DistAlgo process in a distributed system. Each process is uniquely
identified by a `ProcessId` object. Messages exchanged between DistAlgo
processes can be any picklable Python object.

DistAlgo processes can spawn more processes by calling `new`. The process
that called `new` is known as the parent process of the newly spawned
processes. Any DistProcess can send messages to any other DistProcess, given
that it knows the `ProcessId` of the target process. However, only the
parent process can `end` a child process. The terminal is shared between all
processes spawned from that terminal, which includes the stdout, stdin, and
stderr streams.

Concrete subclasses of `DistProcess` must define the methods:

- `setup`: A function that initializes the process-local variables.

- `run`: The entry point of the process. This function defines the
  activities of the process.

Users should not instantiate this class directly, process instances should
be created by calling `new`.
```

Method resolution order:

PastryNode
da.sim.DistProcess

builtins.object

Methods defined here:

__init__(self, procimpl, props)
```
Initialize self.  See help(type(self)) for accurate signature.
```

addMeInRoutingTable(self)
```
Identity map the routing table

The figure 1 in page number 4 presents:
'State of a hypothetical Pastry node with nodeId 10233102, b = 2, and l
= 8. All numbers are in base 4. The top row of the routing table is row
zero. The shaded cell in each row of the routing table shows the
corresponding digit of the present node’s nodeId. The nodeIds in each
entry have been split to show the common prefix with 10233102 - next
digit - rest of nodeId. The associated IP addresses are not shown.'
```

getRelativeDistance(self, coord1, coord2)
```
For scalar procimity metric, we use geographic distance of pastry nodes
for populating the neighborhood set.

Page number 9, section 2.5, paragraph 2 mentions:
'Pastry’s notion of network proximity is based on a scalar proximity
metric, such as the number of IP routing hops or geographic distance.'
```

makeListUnique(self, objSet)
```
Make a list of PastryNodeInfo class objects unique based on nodeid as
nodeid is supposed to be unique.

This is a helper function which is used as python / distalgo uses pass
by value to send class objects as messages due to which usage of inbuild
set() and list() are futile. Hence, this method makes a list unique by
keeping only unique node ids in the list and returns it.

Returns: List of nodes with unique nodeids
```

repairLeafSet(self, failedObj)
```
Repair the leaf set for current node to account for a failed node X

Page number 8, section 2.4, paragraph 5 mentions:
'To replace a failed node in the leaf set, its neighbor in the nodeId
space contacts the live node with the largest index on the side of the
failed node, and asks that node for its leaf table.'
```

repairNeighborSet(self, failedObj)
```
Repair the neighborhood set for current node to account for a failed node X

Page number 8, section 2.4, paragraph 5, line number 4 mentions:
'If a member is not responding, the node asks other members for their
neighborhood tables, checks the distance of each of the newly discovered
nodes, and updates it own neighborhood set accordingly.'
```

repairRoutingTable(self, failedObj, row, col)
```
Function repairs routing table when a node 'X' with object 'failedObj' fails in routing table

Page number 8, section 2.4, paragraph 7 mentions:
'To repair a failed routing table entry R(l,d), a node contacts first
the node referred to by another entry R(l,i), i != d of the same row,
and asks for that node’s entry for R(l,d).  In the event that none of
the entries in row l have a pointer to a live node with the appropriate
prefix, the node next contacts an entry R(l+1, i), i != d, thereby
casting a wider net. This procedure is highly likely to eventually find
an appropriate node if one exists.'
```

route(self, message, dstObj, numHops, tt, tDist)
```
Causes Pastry to route the given message to the node with nodeId
numerically closest to the key, among all live Pastry nodes

Page number 5, section 2.2, Table 1 presents the entire core routing
algorithm:
'
(1) if (L-|L|/2 ≤ D ≤ L|L|/2) {
(2) // D is within range of our leaf set
(3) forward to Li, s.th. |D − Li| is minimal;
(4) } else {
(5) // use the routing table
(6) Let l = shl(D, A);
(7) if (RDl
l = null) {
(8) forward to RDl
l ;
(9) }
(10) else {
(11) // rare case
(12) forward to T ∈ L ∪ R ∪ M, s.th.
(13) shl(T,D) ≥ l,
(14) |T − D| < |A − D|
(15) }
(16) }
```
The entire pseudo code is translated to code to bolster the implementation present in the paper.'

run(self)

This is the driver program for the entire implementation.

Page number 7, section 2.4, paragraph 2 mentions
'When a new node arrives, it needs to initialize its state tables, and then inform other nodes of its presence. We assume the new node knows initially about a nearby Pastry node A, according to the proximity metric, that is already part of the system. Such a node can be located automatically, for instance, using “expanding ring” IP multicast, or be obtained by the system administrator through outside channels.
Let us assume the new node’s nodeId is X. (The assignment of nodeIds is applicationspecific; typically it is computed as the SHA-1 hash of its IP address or its public key). Node X then asks A to route a special “join” message with the key equal to X. Like any message, Pastry routes the join message to the existing node Z whose id is numerically closest to X'
Also, page number 8, paragraph 2 mentions
'Finally, X transmits a copy of its resulting state to each of the nodes found in its neighborhood set, leaf set, and routing table.
Finally, X transmits a copy of its resulting state to each of the nodes found in its neighborhood set, leaf set, and routing table.'

After sending the resulting state, we send joinCompleted request so that the parent() is aware that this node is ready to send and receive messages. We then periodically check if any node needs repair in our neighborhood set as mentioned in page number 9, first paragraph.

'For this purpose, a node attempts to contact each member of the neighborhood set periodically to see if it is still alive.'

Whenever we receive a message ExitProcess, we stop whatever we are doing and terminate the program.

sendState(self, numHops, X)

Sending state to node X

Page number 7, section 2.4, paragraph 4 mentions:
'In response to receiving the “join” request, nodes A, Z, and all nodes encountered on the path from A to Z send their state tables to X.'

setup(self, ctl, boot, b, keySize, L, M, leafSetLen, ti, tack, tn, **rest_4078)

Initialization routine for the DistAlgo process.

Should be overridden by child classes to initialize process states.

shl(self, A, B)

Returns:
The length of the prefix shared among A and B

Page number 5, section 2.2 mentions:
'We begin by defining some notation.
```
Ri l: the entry in the routing table R at column i, 0 ≤ i < 2b and row
      l, 0 ≤ l < 128/b.
Li: the i-th closest nodeId in the leaf set L, −|L|/2 ≤ i ≤ |L|/2, where
    negative/positive indices indicate nodeIds smaller/larger than the
    present nodeId, respectively.
Dl: the value of the l’s digit in the key D.
shl(A, B): the length of the prefix shared among A and B, in digits.'
```

unique_by_key(self, elements, key=None)

This is a helper function to make a list of tuples unique based on a single tuple parameter

updateLeafSet(self, zUpSet, zDownSet, zobj)

Initialize X's leaf set with Z's leaf set

Page number 7, section 2.4, paragraph 5 mentions:
'Moreover, Z has the closest existing nodeId to X, thus its leaf set is the basis for X’s leaf set'

updateNeighborSet(self, neighborSet, obj)

Update X's neighborhood set with A's neighborhood set

Page number 7, section 2.4, paragraph 5 mentions:

```
'Since node A is assumed to be in proximity to the new node X, A’s
neighborhood set to initialize X’s neighborhood set.'
```

updateRoutingTable(self, zRoutingTable, obj)

```
Update the routing table of the current node based on the state
information obtained from other closest nodes present in the routePath

Page number 8, section 2.4, paragraph 5 mentios:
'Next, we consider the routing table, starting at row zero. We consider
the most general case, where the nodeIds of A and X share no common
prefix. Let Ai denote node A’s row of the routing table at level i. Note
that the entries in row zero of the routing table are independent of a
node’s nodeId. Thus, A0 contains appropriate values for X0. Other levels
of A’s routing table are of no use to X, since A’s and X’s ids share no
common prefix. However, appropriate values for X1 can be taken from B1,
where B is the first node encountered along the route from A to Z. To
see this, observe that entries in B1 and X1 share the same prefix,
because X and B have the same first digit in their nodeId.  Similarly, X
obtains appropriate entries for X2 from node C, the next node
encountered along the route from A to Z, and so on.'
```

Methods inherited from da.sim.DistProcess:

__repr__(self)

```
Return repr(self).
```

__str__ = **__repr__**(self)

```
Return repr(self).
```

debug(self, *message, sep=' ')

```
Prints debugging output to the process log.

This is the same as `output` except the message is logged at the
'USRDBG' level.
```

end(self, target, exit_code=1)

```
Terminate the child processes specified by `target`.

`target` can be a process id or a set of process ids, all of which must
be a child process of this process.
```

error(self, *message, sep=' ')

```
Prints error message to the process log.

This is the same as `output` except the message is logged at the
'USRERR' level.
```

exit(self, code=0)

```
Terminates the current process.

`code` specifies the exit code.
```

hanged(self)

```
Hangs the current process.

When a process enters the 'hanged' state, its main logic and all message
handlers will no longer run.
```

incr_logical_clock(self)

```
Increments the logical clock.

For Lamport's clock, this increases the clock value by 1.
```

logical_clock(self)

```
Returns the current value of the logical clock.
```

nameof(self, pid)

```
Returns the process name of `pid`, if any.
```

new(self, pcls, args=None, num=None, at=None, method=None, **props)

```
Creates new DistAlgo processes.

`pcls` specifies the DistAlgo process class. Optional argument `args` is
a list of arguments that is used to call the `setup` method of the child
processes. Optional argument `num` specifies the number of processes to
create on each node. Optional argument `at` specifies the node or nodes
on which the new processes are to be created. If `num` is not specified
then it defaults to one process. If `at` is not specified then it
defaults to the same node as the current process. Optional argument
`method` specifies the type of implementation used to run the new
process(es), and can be one of 'process', in which case the new
processes will be run inside operating system processes, or 'thread' in
```

```
which case the processes will be run inside operating system threads. If
method is not specified then its default value is taken from the
'--default_proc_impl' command line option.

If neither `num` nor `at` is specified, then `new` will return the
process id of child process if successful, or None otherwise. If either
`num` or `at` is specified, then `new` will return a set containing the
process ids of the processes that was successfully created.
```

nodeof(self, pid)

```
Returns the id of the node process that's running `pid`.
```

output(self, *message, sep=' ', level=21)

```
Prints arguments to the process log.

Optional argument 'level' is a positive integer that specifies the
logging level of the message, defaults to 'logging.INFO'(20). Refer to
[https://docs.python.org/3/library/logging.html#levels] for a list of
predefined logging levels.

When the level of the message is equal to or higher than the
configured level of a log handler, the message is logged to that
handler; otherwise, it is ignored. DistAlgo processes are
automatically configured with two log handlers:, one logs to the
console, the other to a log file; the handlers' logging levels are
controlled by command line parameters.
```

parent(self)

```
Returns the parent process id.

The parent process is the process that called `new` to create this
process.
```

resolve(self, name)

```
Returns the process id associated with `name`.
```

send(self, message, to, channel=None, **rest)

```
Send a DistAlgo message.

`message` can be any pickle-able Python object. `to` can be a process id
or a set of process ids.
```

work(self)

```
Waste some random amount of time.

This suspends execution of the process for a period of 0-2 seconds.
```

Class methods inherited from da.sim.DistProcess:

get_config(key, default=None) from builtins.type

```
Returns the configuration value for specified 'key'.
```

Data descriptors inherited from da.sim.DistProcess:

__dict__

```
dictionary for instance variables (if defined)
```

__weakref__

```
list of weak references to the object (if defined)
```

Data and other attributes inherited from da.sim.DistProcess:

AckCommands = [<Command.NewAck: 16>, <Command.EndAck: 15>, <Command.StartAck: 11>, <Command.SetupAck: 12>, <Command.ResolveAck: 17>, <Command.RPCReply: 31>]

class **PastryNodeInfo**(builtins.object)

Methods defined here:

__init__(self, obj, b, keySize)
```
    Initialize self.  See help(type(self)) for accurate signature.
```

changeBase(self, num, base)
```
    Takes a number num in base 10 and converts it to base b

    By default hexdigest is returned from cryptographic function used. We
    convert the number in base 10 to a particular base b. This is useful if
    we change the value of B which is by default 4.
```

getCoordinates(self)
```
    Map the nodes in a 2D grid to determine the relative distance between
    the nodes

    Page number 13, section 3, 3rd paragraph mentions:
    'Each Pastry node is assigned a location in a plane; coordinates in the
    plane are randomly assigned in the range [0, 1000]. Nodes in the
    Internet are not uniformly distributed in a Euclidean space; instead,
    there is a strong clustering of nodes and the triangulation inequality
    doesn’t always hold.'
```

getNodeKey(self, ip, b, keySize)
```
    Get a 128-bit identifier by hashing the ip address

    Page number 3, section 2, paragraph 2 mentions:
    'The nodeId is assigned randomly when a node joins the system. It is
    assumed that nodeIds are generated such that the resulting set of
    nodeIds is uniformly distributed in the 128-bit nodeId space.'
```

getRandomIPAddress(self)
```
    Get random IP Address for a particular node

    Page number 3, section 2, paragraph 2 mentions:
    'For instance, nodeIds could be generated by computing a cryptographic
    hash of the node’s public key or its IP address.'
```

Data descriptors defined here:

__dict__
```
    dictionary for instance variables (if defined)
```

__weakref__
```
    list of weak references to the object (if defined)
```

Functions

sqrt(...)
```
    sqrt(x)

    Return the square root of x.
```

Data

INF = 999999999999999999...9999999999999999999
PatternExpr_1739 = (='acknowledgement',=_BoundPattern1742_)
PatternExpr_1745 = nextNode
PatternExpr_1872 = (='acknowledgement',=_BoundPattern1875_)
PatternExpr_1878 = =_BoundPattern1893_
PatternExpr_2298 = (='acknowledgement',=_BoundPattern2301_)
PatternExpr_2304 = nextNode
PatternExpr_2516 = (='LeafSet',xUpLeafSet,xDownLeafSet,=_BoundPattern2523_)
PatternExpr_2526 = =_BoundPattern2529_
PatternExpr_258 = (='getRandomNodes',numNodes)
PatternExpr_2581 = (='NodeIsAlive',=_BoundPattern2584_)
PatternExpr_2587 = =_BoundPattern2590_
PatternExpr_265 = X
PatternExpr_2757 = (='LeafSet',xUpLeafSet,xDownLeafSet,=_BoundPattern2762_)
PatternExpr_2765 = =_BoundPattern2768_
PatternExpr_2817 = (='NodeIsAlive',=_BoundPattern2820_)
PatternExpr_2823 = =_BoundPattern2826_
PatternExpr_2977 = (='NeighborhoodSet',xNeighborhoodSet,=_BoundPattern2982_)
PatternExpr_2985 = =_BoundPattern2988_
PatternExpr_3025 = (='NodeIsAlive',=_BoundPattern3028_)
PatternExpr_3031 = =_BoundPattern3034_
PatternExpr_3163 = (='RoutingTableEntry',xRTEntry,=_BoundPattern3168_)
PatternExpr_3171 = =_BoundPattern3174_
PatternExpr_3211 = (='NodeIsAlive',=_BoundPattern3214_)
PatternExpr_3217 = =_BoundPattern3220_
PatternExpr_3253 = (='getNeighborhoodSet',t)
PatternExpr_326 = (='NodeIsAlive',=_BoundPattern329_)
PatternExpr_3260 = X
PatternExpr_3276 = (='getLeafset',t)
PatternExpr_3283 = X
PatternExpr_3302 = (='getRoutingTableEntry',row,col,t)
PatternExpr_3313 = X
PatternExpr_3334 = (='isNodeAlive',t)
PatternExpr_3341 = X
PatternExpr_3354 = (='stateTables',zRoutingTable,zNeighborhoodSet,obj,numHops)
PatternExpr_3367 = N
PatternExpr_3462 = (='resultingStateTables',downSet,upSet,xRoutingTable,xNeighborhoodSet,obj)
PatternExpr_3477 = X
PatternExpr_3501 = (='join',obj,numHops,_,_,_)
PatternExpr_3513 = X
PatternExpr_3552 = (='lookup',obj,numHops,t1,t2,tDist)
PatternExpr_3567 = X
PatternExpr_3588 = (='getHopCount',src,dst)
PatternExpr_3597 = X
PatternExpr_3620 = (='acknowledgement',=_BoundPattern3623_)
PatternExpr_3626 = =_BoundPattern3629_
PatternExpr_3660 = (='done_lookup',numHops,=_BoundPattern3665_,tDist)
PatternExpr_370 = (='getClosestNeighbor',data)
PatternExpr_3743 = (='closestNode',A,nid)
PatternExpr_377 = X
PatternExpr_3816 = (='done_join',upSet,downSet,maxHops,obj,tDist)
PatternExpr_3858 = (='stateTables',_,_,_,_)
PatternExpr_3886 = (='stateTables',_,_,o,_)
PatternExpr_4014 = (='NodeIsAlive',=_BoundPattern4017_)
PatternExpr_4020 = =_BoundPattern4023_
PatternExpr_4050 = ='ExitProcess'
PatternExpr_472 = ='ExitProcess'