An Algorithm for Distributed Data Grid and Distributed Task Grid: Across Internet in a SuperCluster environment

This paper describes an algorithm for distributed data and task grid supercluster across multiple subnets over the internet. Appendix of the paper has a reference implementation of same in a pure java including the GitHub Source URL.

Definition

Let me first describe the terms.

Distributed Data Grid

A distributed data grid is data is stored in a cluster of machines (in this case, java virtual machines) and seamlessly available to all machines. If one machine adds a data property (a name/value pair), it is immediately available to other machines. Similarly, if one machine removes a property, it will be immediately unavailable to other machines. Moreover, if a new node joins the cluster environment, it gets access to all the data elements immediately.

Distributed Task Grid

A distributed task grid is a clustered grid computing environment. Whenever a task is submitted in the task grid, the cluster setup will automatically choose a designated machine to execute the task. Tasks are always executed asynchronously with or without future result reference, which is available to the caller, to retrieve the result later when task execution is completed.

Cluster Service

Cluster service is the heart of this distributed environment. Each cluster nodes have sender and receiver processes for sending and receiving cluster messages. Therefore. Each node must have its TCP/IP address so that it can receive message there. This address is nothing but the combination of IP Address and TCP Port. Hence each node must have a configuration to specify which TCP Port it should use as the address, the IP defaults to the hosting PCs IP (if there are multiple IPs or multiple IP versions like 4 and 6, it defaults the first available network interface and user-preferred version of the host machine, e.g. JVM preference).

Discovery during start-up

Moreover, the node must announce its availability upon startup, so that other existing nodes become aware of its existing. This algorithm works in auto-discovery way – so that there is no pre-set list of nodes in the cluster – nodes discover each other when activated. The algorithm for auto-discovery varies based on the network setup as described below.

In a Network where UDP Multicasting/Broadcasting is allowed

I am not going in detail on basic networking on how multicasting and broadcasting work and how it is set up within a network. I am assuming, the reader has enough knowledge about the same (or, can search the internet for the same).

If cluster nodes are started within a network where UDP Broadcasting/Multicasting is enabled, it used UDP broadcasting to announce its activation within the network. Already running nodes will be receiving this broadcast and add this new node as the cluster member.

Alternatively, we can use multicasting instead of broadcasting. Multicasting will allow creating multicast groups so that we can create multiple independent clusters using different multicast address. For this to work, the cluster node must be started with a 'group' parameter which will specify the multicast address to join (Multicast address must have an IP range of 224.0.0.0 to 239.255.255.255).

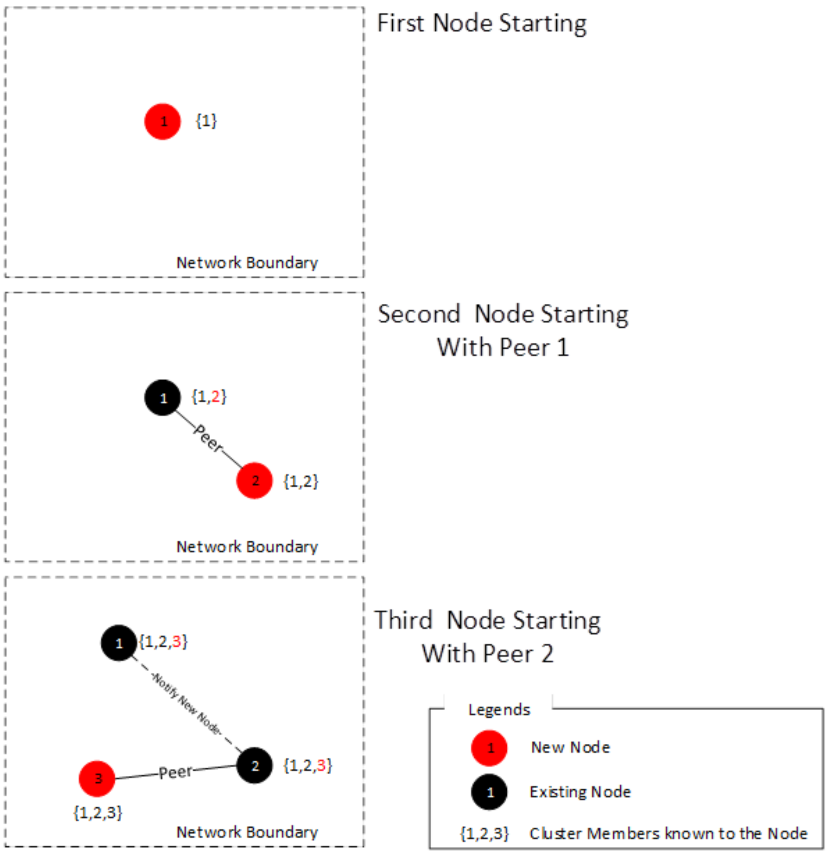
In a Network where UDP Multicasting/Broadcasting is blocked

Typically, across subnets, UDP Multicast/Broadcast is blocked by default router setting (to prevent DDoS attack). Sometimes. The organizational policy prevents UDP multicast/broadcast within same network as well (Except maybe only port 67/68 for DHCP discovery).

In this scenario, a node must be started with a parameter specifying the address of any peer node of the cluster. It works in the following way:

1. If this is the first node in the cluster, start it without any parameter.
2. Otherwise, start the node with a parameter specifying the address (IP and Port) of any currently active node as it's peer node. During start-up, the node will enquire its peer node for all the members of the network and add all the existing nodes in its member list.

Using this algorithm, the cluster grows as the nodes are added. The following picture depicts the algorithm.



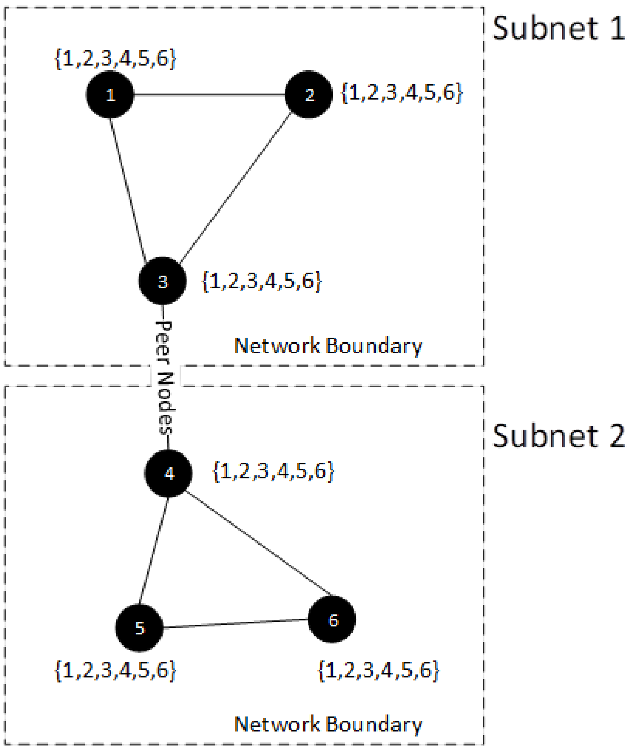
Picture 1: The algorithm of discovering node using peer node mode

In a hybrid Network where subnets support UDP Multicasting/Broadcasting but across subnets UDP Multicasting/Broadcasting is blocked

This is a typical case for the internet. We have multiple individual LANs where UDP Multicasting/Broadcasting is allowed, but across LANs, UDP broadcasting is blocked. To support this type of network is essential for the supercluster setup.

In this scenario, nodes within the subnet will be started using UDB Broadcast/Multicast mode discovery. So there will be multiple sets of cluster nodes in each subnet. One of the nodes in this set must be started using a parameter specifying a peer node address of another node in another subnet.

The algorithm will now be able to discover each node across the networks. The following picture depicts this scenario.



Picture 2: The algorithm of discovering node across subnets

Please note, peer node is used just during start-up. Post-start-up every node are known to every node (discovered via peer node during startup). So, if peer node(s) crashes/stopped afterward, every node is still connected to each other and keep on discovering each other and each other's member via heartbeat services.

Heartbeat service – continuous discovery

Each node will send a heartbeat message to all participating nodes (using TCP), irrespective of they are currently running or dead. There are three possible responses it can get :

- The node is alive and running as a leader (leader)
- The node is alive and running as a normal node (member)
- No Response (timed out)

This service runs periodically and updates the running member list according to the response received. The heartbeat response also carries the member list attached to the list, so that each node can discover all nodes in the network.

Leader selection service

If heartbeat service finds there is no leader present in a particular run, it starts this service to assign a leader for the cluster. This service uses the following algorithm to select a leader:

- Each node sleeps for a variable amount of time, none of the nodes have exactly the same amount of time. This is achieved by assigning a seep time = <node position> x <fixed time>. For example, the first node (in the order of configuration position) will sleep for 1 second, the 2nd node will sleep for 2 seconds and so on.
- After waking up from sleep, each node will check if there is a leader present in the cluster. If still there is no leader present, it will assign itself as the leader. For example, the first node will wake up after 1 second and will see that there is no leader, it will mark itself as the leader. The second node, after waking up after 2 seconds will find there is already a leader (first node). The checking of a leader happens after the fixed time set before. So, if the fixed time is 1 second, every node will check if there is a leader every second, but assign itself as a leader only after the number of loops is equal to the position of the node. So for 1 second fixed time, the leader will be assigned after 1 second and every node will find the leader in 2 seconds time and happily exit the loop.

Cluster Messaging

Each cluster node listens on a predefined TCP port for messages sent by other nodes. There are three categories of messages:

Administrative messages

Heartbeat message and sync messages are examples of an administrative message. Heartbeat message determines the status of a node. Sync message is sent to the leader node when a node is starting up (if it is not the first node to start up). Sync message syncs the data and task grid to initialize its data and task executors.

Operational Messages

These messages are operational nature and generally broadcasting in nature. This instructs the other node to perform an operation. For example, if one node adds a data, it is broadcasted to other node so that the same operation is repeated on other nodes to be data in sync. Example of a non-broadcasting message is instructing a node to execute a task.

Retrieve a zipped file

This is a special message where one node can send a message to a particular node to receive one or more file from the remote directory. The remote node zips all matching file from the requested directory sends the zipped file as a stream. This is particularly useful to share log files between the nodes.

Security Concept

This part explains how security aspects are implemented in the cluster node to node communication. The codebase has already this changes built in. Security is built-in into the solution, thus can not be switched off. The API construct does not change, therefore developer using this library do not have to do anything extra to enable the security.

There are two aspects of the security that we should be taken care.

Message Security

All communications between the cluster nodes must be secured, so that no body can read the messages in transit. Moreover, the sender can encrypt the message such a way that only the intended recipient will be able to decrypt it. Therefore we will be using asymmetric encryption mechanism (RSA Key pair) – where sender will use receiver’s public key to encrypt the message. Therefore, only receiver having the private key can decrypt the message.

Node Identity

Only having data-in-transit encryption is not enough. Any rogue agent can construct a message, having access to the public key. The message may carry trojan horse like executable task, which upon execution at receiver node (or any other node based on task distribution policy) may expose the private key of the nodes. Therefore, in addition to asymmetric encryption, messages must be digitally signed, so that receiver can ensure that the message is received from a trusted party. Here, we will be using again asymmetric key pair (DSA Key Pair), sender will sign the message using it’s private key, receiver will use the sender’s public key to verify the message.

Security Implementation

The security is implemented using three distinct steps. Step one is executed when a node is started, step two is how the public keys are exchanged, whereas step three is the actual communication. For step one we would examine how security credentials are established. For step two we will examine how the keys are exchanged during nodes discovery. Step three is really about the security protocol which uses theses security credentials.

Node Start Up

Security credentials are initialized during node start up. Each node generates two key pairs:

1. RSA public and private key pair
2. DSA public and private key pair

Each node keeps its private keys in its memory, and it is not sharable. This is achieved using ‘transient’ field of the member object – therefore it is not part of the message bundling.

On the other hand, public keys are shareable and sent to all nodes when it shares the node details with other nodes using multicast/broadcast or peer-discovery method.

Node Discovery and Heartbeats

As we have seen earlier in this paper, during start-up every nodes announces its presence in the network. This done either using multicast/broadcast message or via a peer node. While announcing its presence in the network, the node also shares the two public keys with all. In that way each node becomes aware of all the public keys of other nodes.

Security Protocol

Each message is encoded before sending to a node. While sending, the sender knows, public keys of the sender and receiver. There is a unique case, when a node is announcing its presence to other. In that situation, sender is not aware of the public keys of the receiver. Also, receiver is not aware of the sender's public key. Only this message is sent without any security. This is really a handshake message where the presence of the new node is being announced and public keys are exchanged. But all subsequence messages will be encrypted and signed. Therefore, we must put a marker byte in beginning of each message to distinguish an unsecured message with a secured message. We have put 0x00 as marker for unsecured message and 0zff as marker for secured message.

Unsecure message construct is very simple – as shown below:

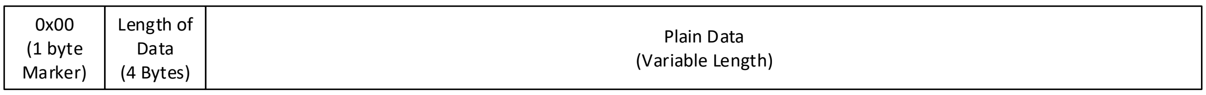


Figure 1: Unsecured Message

The following section explains the secured message.

Secured Message

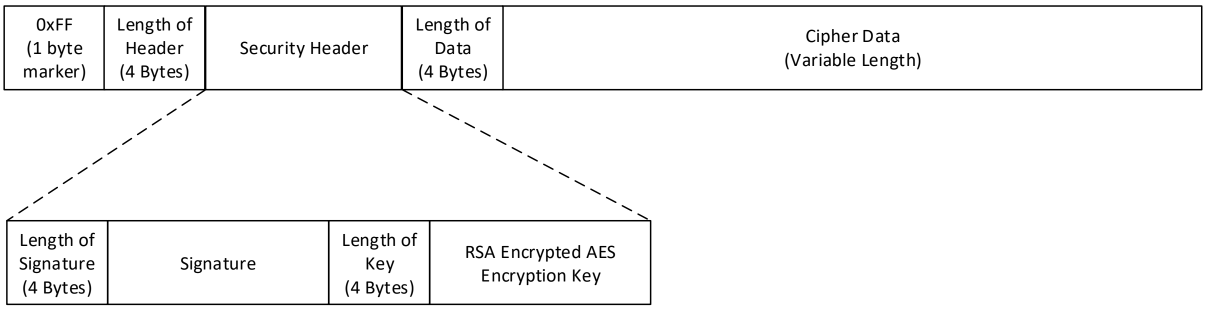

0xFF (1 byte marker)
Length of Header (4 Bytes)
Security Header
Length of Data (4 Bytes)
Cipher Data (Variable Length)
Length of Signature (4 Bytes)
Signature
Length of Key (4 Bytes)
RSA Encrypted AES Encryption Key

Figure 2: secured message

The steps for constructing a secured message is as follows:

1. An AES 256-bit key is generated randomly. To improve performance, a generated key is cached. But, key is expired after 2 minutes, so that key gets rotate every 2 minutes for added security.

2. Plain data is encrypted using the AES key and a cipher data is created.

3. The AES is key is encrypted using the RSA public key of the receiver.

4. The encrypted AES key and cipher data is packed together (each block precedes the length of the block, as shown in picture).

5. This new block is signed using the sender's DSA private key. The signature is added to the key-data block.

The steps to decode is reverse of above.

1. Signature is extracted.
2. Signature is verified using the sender's DSA public key and the key-data block.
3. If verification failed, decoding is failed.
4. AES Key is decrypted using receiver's RSA private key.
5. The cipher data is decrypted using the AES key.

Appendix: Java Implementation

The code for same is available in following GIT Repository:

https://github.com/tuhinsengupta/cluster

Java API: ClusterService (Cluster Service)

Constructors	
Constructor and Description	
private ClusterService(ClusterConfig config) Creates the Cluster Service with the given configuration. This is a singleton class, hence the constructor is private. Use getInstance(config) API to create a Cluster Service.	
Methods	
Modifier and Type	**Method and Description**
void	**clear**() Removes all of the data from this map in the grid.
Object	**clone**() Returns a shallow copy of this DistributedMap instance: the keys and values themselves are not

	cloned. Will use the same name reference, hence any add remove operation will affect the original Map. Not recommended to use.
boolean	**containsKey**(**Object** key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(**Object** value) Returns true if this map maps one or more keys to the specified value.
Set<**Map.Entry**<**K**, **V**>>	**entrySet**() Returns a **Set** view of the mappings contained in this map.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<**K**>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map. The value will be copied across all node in the cluster.
void	**putAll**(**Map**<? extends **K**,? extends **V**> m) Copies all of the mappings from the specified map to this map. The values will be copied across all node in the cluster.
V	**remove**(**Object** key) Removes the mapping for the specified key from this

	map if present. The value will be removed from all node in the cluster.
int	**size**() Returns the number of key-value mappings in this map.
Collection<**V**>	**values**() Returns a **Collection** view of the values contained in this map.

Usage Examples

Example 1

Start ClusterService in default mode. In this mode, the node will be using UDP Broadcast to discover other nodes and will start in a next available port of the host machine.

```
//Declare the ClusterService
ClusterService service;
//Start the service
service = ClusterService.getInstance(new ClusterConfig());
//Do your work using ClusterService
//Stop the service
service.stop();
```

Example 2

Start ClusterService in specified port 9090 and use UDP Multicast group "239.1.2.3." to discover other nodes.

```
//Declare the ClusterService
ClusterService service;
//Start the service with port 9090 and Multicast group "239.1.2.3"
service = ClusterService.getInstance(new ClusterConfig()
                                     .setPort(9090)
                                     .setMulticastGroup("239.1.2.3")
                                    );
//Do your work using ClusterService
//Stop the service
service.stop();
```

Example 3

Start ClusterService in specified port 9092 and join to the cluster (created using Example 2 above) using peer node, IP=12.45.23.10 and Port=9090.

```
//Declare the ClusterService
ClusterService service;
//Start the service with port 9090 and Multicast group "239.1.2.3"
service = ClusterService.getInstance(new ClusterConfig()
                                     .setPort(9090)
                                     .setMulticastGroup("239.1.2.3")
                                     .setPeerNode("12.45.23.10:9090")
                                    );
//Do your work using ClusterService
//Stop the service
service.stop();
```

Java API: ClusterConfig (Cluster Service Configuration)

Constructors	
Constructor and Description	
public ClusterConfig() Creates the Cluster Service Configuration with the default configuration.	
Methods	
Modifier and Type	**Method and Description**
ClusterConfig	**setPort**(int port) Set the port of the node. Defaults to zero - thus, Java allocates the next available port of the host machine. Returns the current ClusterConfig so that options can be set in the chain.
ClusterConfig	**setMulticastGroup**(String group) Set the multicast group address. Valid values are 224.0.0.0 to 239.255.255.255. If not set, the UDP broadcast is used instead. Returns the current ClusterConfig so that options can be set in the chain.
ClusterConfig	**setMulticastPort**(int port) Set the multicast port for listening multicast messages. The default value is 8888. Returns the current ClusterConfig so that options can be set in the chain.
ClusterConfig	**setWeight**(int weight) Set the node weight, used for weight-based task distribution. The default value is 1. Returns the current ClusterConfig so that options can be set

	in the chain.
ClusterConfig	**setSocketBacklog**(int backlog) Sets the maximum length of the queue of incoming connections.The default value is 50. Returns the current ClusterConfig so that options can be set in the chain.
ClusterConfig	**setHeartBeatInterval**(int interval) Set the interval in milliseconds for the heartbeat messages between nodes. The default value is 1000 (1 Second). Returns the current ClusterConfig so that options can be set in the chain.
ClusterConfig	**setMaxWait**(int wait) Set the thread wait time in minutes while terminating threads. The default value is 30 minutes. Returns the current ClusterConfig so that options can be set in the chain.
ClusterConfig	**setNetworkTimeout**(int timeout) Set the multicast group address. Valid values are 224.0.0.0 to 239.255.255.255. If not set, the UDP broadcast is used instead. Returns the current ClusterConfig so that options can be set in the chain.
ClusterConfig	**setPeerNode**(int peer) Sets the peer node. The format of the peer node address should be <host>:<port>. Returns the current ClusterConfig so that options can be set in the chain.

Usage Example

```
ClusterConfig config = new ClusterConfig()
                        .setPort(9090)                    //Set port 9090
                        .setWeight(2)                     //Set node weight
2
                        .setMulticastGroup("239.1.2.3")   //Set multicast g
roup address 239.1.2.3
                        .setMulticastPort(8080)           //Set multicast l
isten port to 8080
                        .setSocketBacklog(100)            //Set socket inco
ming request queue size to 100
                        .setHeartBeatInterval(500)        //Set Heart beat
interval to 500 ms
                        .setMaxWait(10)                   //Set termination
wait time to 10 minutes
                        .setNetworkTimeout(3000)          //Set network tim
eout to 3000 ms = 3 seconds
                        .setPeerNode("12.34.56.123:9090") //Set peer node t
o 12.34.56.123:9090
;
```

Java API: DistributedMap<K, V> (Distributed Data Grid)

Java has an existing Interface called Map<K, V>. The Distributed Data Grid implements this standard java interface, hence the API contract is exactly the same as Map<K, V> interface. The implementation class name is DistributedMap<K, V> and the API contract is defined below:

Constructors	
Constructor and Description	
DistributedMap(ClusterService service, String name) Creates the distributed data grid on the Cluster Service ‘service’ with the name ‘name’.	
Methods	
Modifier and Type	**Method and Description**
void	**clear**() Removes all of the data from this map in the grid.
Object	**clone**()

	Returns a shallow copy of this DistributedMap instance: the keys and values themselves are not cloned. Will use the same name reference, hence any add remove operation will affect the original Map. Not recommended to use.
boolean	**containsKey**(**Object** key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(**Object** value) Returns true if this map maps one or more keys to the specified value.
Set<**Map.Entry**<**K**, **V**>>	**entrySet**() Returns a **Set** view of the mappings contained in this map.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<**K**>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map. The value will be copied across all node in the cluster.
void	**putAll**(**Map**<? extends **K**,? extends **V**> m) Copies all of the mappings from the specified map to this map. The values will be copied across all node in the cluster.

V	**remove**(**Object** key) Removes the mapping for the specified key from this map if present. The value will be removed from all node in the cluster.
int	**size**() Returns the number of key-value mappings in this map.
Collection<**V**>	**values**() Returns a **Collection** view of the values contained in this map.

Usage Example:

```
DistributedMap<String, String> map = new DistributedMap<String, String>(ser
vice, "myMap");
map.put("key1", "value1");
map.get("key1");
```

This will create a Map which shared across all the cluster nodes defined by the ‘service’.

So if ‘node1’ creates a map, that will be immediately available to the ‘node2’, with get() call.

Java API: DistributedThreadPool (Distributed Service Grid)

Java has an existing Interface called ExecutorService<K, V>. The Distribute Service Grid implements this standard java interface, hence the API contract is exactly the same as ExecutorService interface. The implementation class name is DistributedThreadPool and the API contract is defined below:

Constructors
Constructor and Description
DistributedThreadPool(ClusterService service, TaskDistributingPolicy policy, int threadPoolSize) Creates the distributed service grid on the Cluster Service 'service' with policy and thread pool size specified. The distribution policy could be : TaskDistributingPolicy.*Random:* tasks are distributed to the random node randomly selected among the running nodes. TaskDistributingPolicy.*RoundRobin:* tasks are distributed among the running nodes with round-robin way. TaskDistributingPolicy.*WeightedRoundRobin:* tasks are distributed among the running nodes with round-robin way, but based on the weight ratio of the nodes. For example, if there are 2 nodes running, first node weight is 2 and the second node weight is 1; then first two tasks will be submitted on node 1, next 1 task will be submitted on node 1 and so on.
DistributedThreadPool(ClusterService service, int threadPoolSize) Creates the distributed data grid on the Cluster Service 'service' with default round-robin distribution policy and thread pool size specified.
DistributedThreadPool(ClusterService service, TaskDistributingPolicy policy) Creates the distributed data grid on the Cluster Service 'service' with a specified distribution policy and default thread pool size of 5.
DistributedThreadPool(ClusterService service) Creates the distributed data grid on the Cluster Service 'service' with default round-robin distribution policy and default thread pool size of 5.

Methods	
Modifier and Type	**Method and Description**
boolean	**awaitTermination**(long timeout, **TimeUnit** unit) Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
<T> **List**<**Future**<T>>	**invokeAll**(**Collection**<? extends **Callable**<T>> tasks) Executes the given tasks, returning a list of Futures holding their status and results when all complete.
<T> **List**<**Future**<T>>	**invokeAll**(**Collection**<? extends **Callable**<T>> tasks, long timeout, **TimeUnit** unit) Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.
<T> T	**invokeAny**(**Collection**<? extends **Callable**<T>> tasks) Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.
<T> T	**invokeAny**(**Collection**<? extends **Callable**<T>> tasks, long timeout, **TimeUnit** unit) Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.
boolean	**isShutdown**() Returns true if this executor has been shut down.
boolean	**isTerminated**() Returns true if all tasks have completed following shut down.

void	**shutdown**() Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
List<**Runnable**>	**shutdownNow**() Attempts to stop all actively executing tasks halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
<T> **Future**<T>	**submit**(**Callable**<T> task) Submits a value-returning task for execution and returns a Future representing the pending results of the task.
Future<?>	**submit**(**Runnable** task) Submits a Runnable task for execution and returns a Future representing that task.
<T> **Future**<T>	**submit**(**Runnable** task, T result) Submits a Runnable task for execution and returns a Future representing that task.

Usage Example:

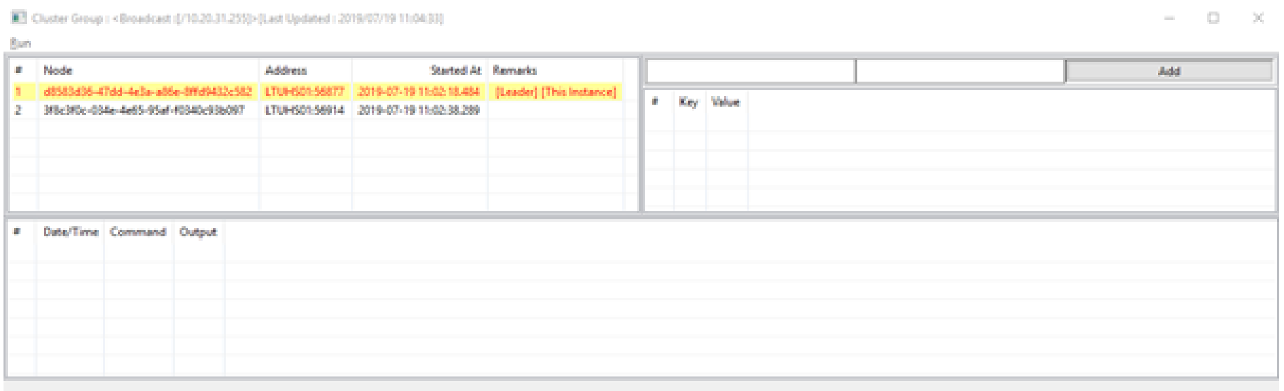
```
DistributedThreadPool pool = new DistributedThreadPool(service, TaskDistrib
utingPolicy.RoundRobin, 20);
pool.submit(new Runnable() {
                        @Override
                        publicvoid run() {
                              System.out.println("Hello World!");
                        }
                  });
```

This will submit the task of printing “Hello World!” to the next available node of the cluster of nodes defined by ‘service’.

Cluster Demo Screenshot

I have also build a GUI based demo for the same. The starting class is *org.tuhin.app.ClusterDemo*

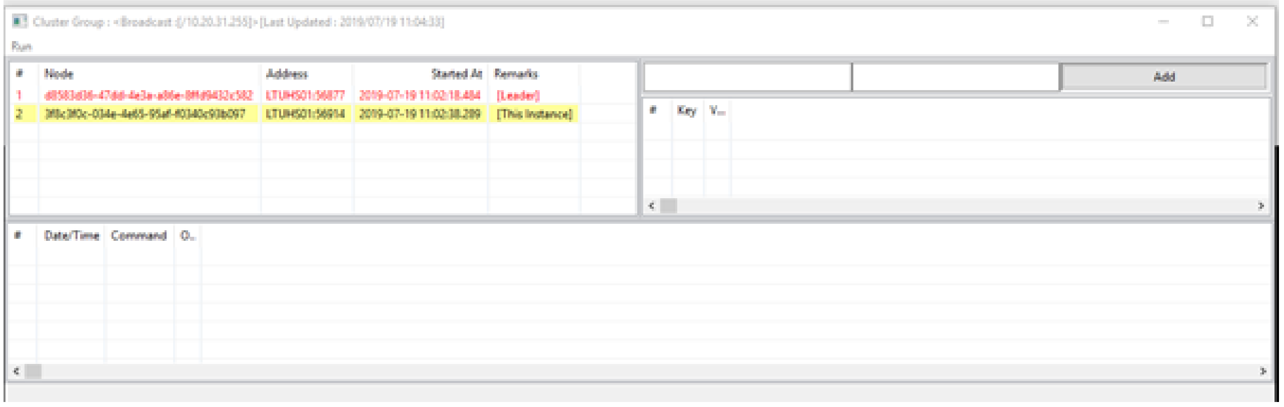
Here is some screenshot for same.



Cluster Group : <Broadcast :[/10.20.31.255]>[Last Updated : 2019/07/19 11:04:33]

Run

#	Node	Address	Started At	Remarks
1	d8583d36-47dd-4e3a-a86e-8ffd9432c582	LTUHS01:56877	2019-07-19 11:02:18.484	[Leader] [This Instance]
2	3f8c3f0c-034e-4e65-95af-f0340c93b097	LTUHS01:56914	2019-07-19 11:02:38.289	



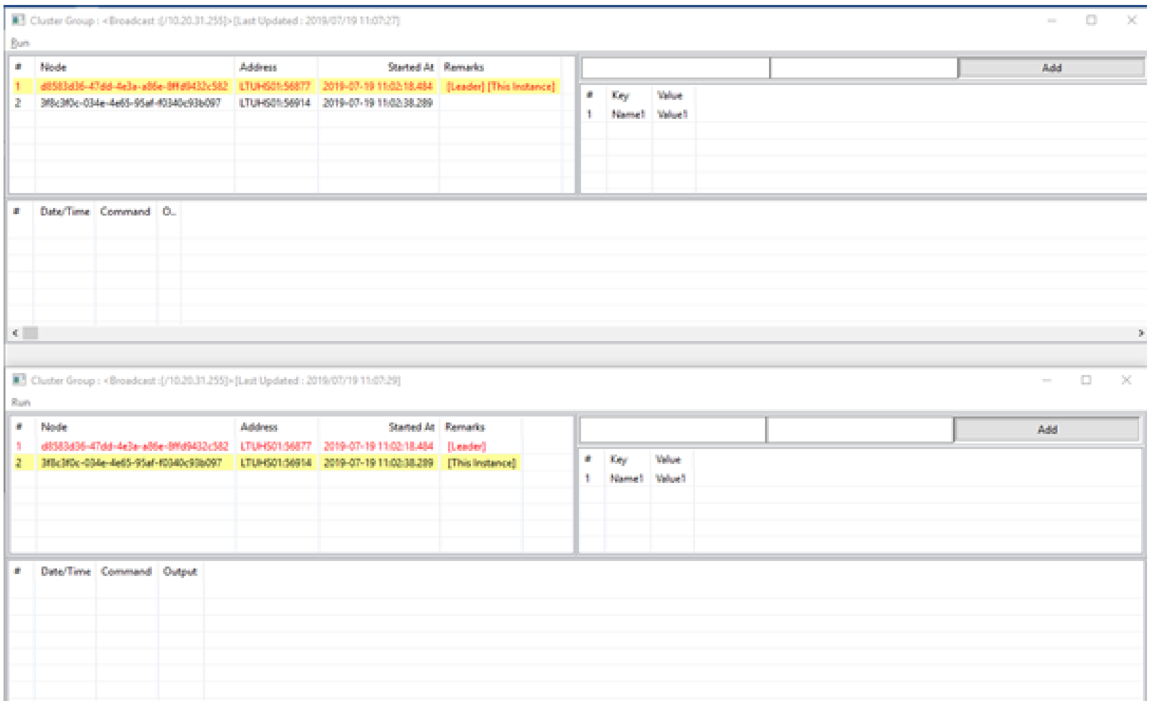
Cluster Group : <Broadcast :[/10.20.31.255]>[Last Updated : 2019/07/19 11:04:33]

Run

#	Node	Address	Started At	Remarks
1	d8583d36-47dd-4e3a-a86e-8ffd9432c582	LTUHS01:56877	2019-07-19 11:02:18.484	[Leader]
2	3f8c3f0c-034e-4e65-95af-f0340c93b097	LTUHS01:56914	2019-07-19 11:02:38.289	[This Instance]

1. Two Nodes Running

- Yellow highlight: the current node
- Red Text: Leader Node



Cluster Group : <Broadcast :[/10.20.31.255]>[Last Updated : 2019/07/19 11:07:27]

Run

#	Node	Address	Started At	Remarks
1	d8583d36-47dd-4e3a-a86e-8ffd9432c582	LTUHS01:56877	2019-07-19 11:02:18.484	[Leader] [This Instance]
2	3f8c3f0c-034e-4e65-95af-f0340c93b097	LTUHS01:56914	2019-07-19 11:02:38.289	

#	Key	Value
1	Name1	Value1

#	Date/Time	Command	O..

Cluster Group : <Broadcast :[/10.20.31.255]>[Last Updated : 2019/07/19 11:07:29]

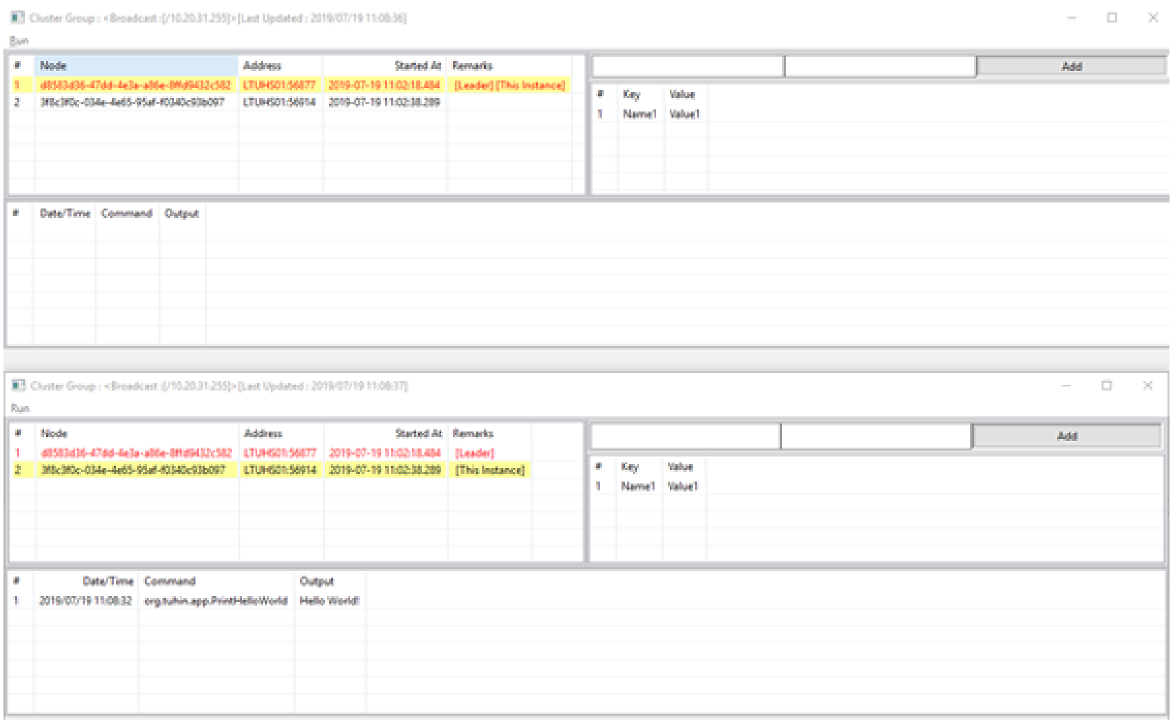
Run

#	Node	Address	Started At	Remarks
1	d8583d36-47dd-4e3a-a86e-8ffd9432c582	LTUHS01:56877	2019-07-19 11:02:18.484	[Leader]
2	3f8c3f0c-034e-4e65-95af-f0340c93b097	LTUHS01:56914	2019-07-19 11:02:38.289	[This Instance]

#	Key	Value
1	Name1	Value1

#	Date/Time	Command	Output

2. Added a Name/Value in one Node – appeared in another node instantly.



Cluster Group : <Broadcast :[/10.20.31.255]> [Last Updated : 2019/07/19 11:08:36]

Run

#	Node	Address	Started At	Remarks
1	d8583d36-47dd-4e3a-a86e-8ffd9432c582	LTUHS01:56877	2019-07-19 11:02:18.484	[Leader] [This Instance]
2	3f8c3f0c-034e-4e65-95af-f0340c93b097	LTUHS01:56914	2019-07-19 11:02:38.289	

#	Key	Value
1	Name1	Value1

#	Date/Time	Command	Output

Cluster Group : <Broadcast :[/10.20.31.255]> [Last Updated : 2019/07/19 11:08:37]

Run

#	Node	Address	Started At	Remarks
1	d8583d36-47dd-4e3a-a86e-8ffd9432c582	LTUHS01:56877	2019-07-19 11:02:18.484	[Leader]
2	3f8c3f0c-034e-4e65-95af-f0340c93b097	LTUHS01:56914	2019-07-19 11:02:38.289	[This Instance]

#	Key	Value
1	Name1	Value1

#	Date/Time	Command	Output
1	2019/07/19 11:08:32	org.tuhin.app.PrintHelloWorld	Hello World!

3. Ran ‘Hello World’ in the upper node – it ran in below node.