**API Documentation**

Version 0.0.02

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 07-15-2017 | 0.0.01 | First API document | Anindya Gautam |
| 08-03-2017 | 0.0.02 | Second API document | Yunpeng Xu |

**Formal Approval**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Issues** | **Approved by** |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Introduction 4](#_Toc489806402)

[1.1 Purpose 4](#_Toc489806403)

[1.2 Definitions, Acronyms, Abbreviations 4](#_Toc489806404)

[2. User Guide 5](#_Toc489806405)

[3. Application Programing Interface (API) offered by SCN Library 7](#_Toc489806406)

[4. SCN Services 17](#_Toc489806407)

[5. Detailed Sample Use Case 17](#_Toc489806408)

[6. Guidelines of Wrapping 20](#_Toc489806409)

# **Introduction**

The Turtlebot project aims at building a framework to support configuration/reconfiguration for mobile robots at run time.

## Purpose

The purpose of this document is to provide the user of the reconfiguration framework with a guide on how to use the APIs provided by the framework library and an overview of all the APIs provided by the framework library.

## Definitions, Acronyms, Abbreviations

|  |  |  |
| --- | --- | --- |
| **DAA** | **Term** | **Definition** |
| ROS | Robot Operating System |  |
| Bot | Robot |  |
| TBD | To Be Discussed |  |
| SCN | System Control Node |  |
| API | Application Programing Interface |  |

# **User Guide**

1. **Initialization:**

For initializing a node within the reconfiguration framework, ros::NodeHandle has been wrapped using ros::SCNNodeHandle, so now when an application developer is writing the code for a node, he shall have to write the statement:

ros::SCNNodeHandle n;

instead of the statement:

ros::NodeHandle n;

Our framework also provides wrapper of the original ros::init with our framework API scnInit(int &argc, char\*\* argv, const std::string &name, uint32\_t options, saveStateRoutine saveStateCb, reconModeRoutine reconModeCb, loadStateRoutine loadStateCb).

**Parameters**

|  |  |
| --- | --- |
| **argc** | Count of argument in argv |
| **argv** | Argument list |
| **name** | Name of the node |
| **options** | Init with specific options |
| **saveStateRoutine** | Callback to save state of the node |
| **reconModeRoutine** | Callback to be called when entering/exiting recon mode |
| **loadStateRoutine** | Callback to load state of the node |

**Returns**

No return parameters

1. **Publishers**:
   1. **Advertising a publisher:**

An example of the existing ROS API for this functionality is as follows:

ros::Publisher chatter\_pub = n.advertise<std\_msgs::String>("chatter", 1000);

For nodes using the reconfiguration framework, the following API needs to be called instead:

ros::SCNPublisher chatter\_pub = n. advertise<std\_msgs::String>(“talker”, "chatter", 1000);

1. **Subscribers:**
   1. **Subscribing to a topic:**

An example of the existing ROS API for this functionality is as follows:

ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

For nodes using the reconfiguration framework, the following API needs to be called instead:

ros:: SCNSubscriber sub = n.subscribe(“listener”, "chatter", 1000, chatterCallback);

1. **Servers:**
   1. **Advertising a service:**

An example of the existing ROS API for this functionality is as follows:

ros::ServiceServer service = n.advertiseService("add\_two\_ints", add);

For nodes using the reconfiguration framework, the following API needs to be called instead:

ros::SCNServiceServerservice = n.advertiseService("add\_two\_ints\_server", "add\_two\_ints", add);

1. **Clients:**
   1. **Becoming the client for a service:**

An example of the existing ROS API for this functionality is as follows:

ros::ServiceClient client = n.serviceClient<beginner\_tutorials::AddTwoInts>("add\_two\_ints");

For nodes using the reconfiguration framework, the following API needs to be called instead:

ros::ServiceClient client = n.SCNServiceClient<beginner\_tutorials::AddTwoInts>( "add\_two\_ints\_client", "add\_two\_ints”);

* 1. **Calling a service**

An example of the existing ROS API for this functionality is as follows:

if (client.call(srv))

{

…

}

For nodes using the reconfiguration framework, this API remains the same:

if (client.call(srv))

{

…

}

# **Application Programing Interface (API) offered by SCN Library**

* 1. Publishers:

1) template<class M> SCNPublisher advertise(const std::string &node\_name,

const std::string &topic\_published,

uint32\_t queue\_size,

bool latch=false)

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **topic** | Topic to advertise on |
| **queue\_size** | Maximum number of outgoing messages to be queued for delivery to subscribers |
| **latch** | (optional) If true, the last message published on this topic will be saved and sent to new subscribers when they connect |

**Returns**

On success, a SCNPublisher that, when it goes out of scope, will automatically release a reference on this advertisement. On failure, an empty SCNPublisher.

2) template<class M> SCNPublisher advertise(const std::string &node\_name,

const std::string& topic\_published,

uint32\_t queue\_size,

const SubscriberStatusCallback& connect\_cb,

const SubscriberStatusCallback& disconnect\_cb = SubscriberStatusCallback(),

const VoidConstPtr& tracked\_object = VoidConstPtr(),

bool latch = false)

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **topic** | Topic to advertise on |
| **queue\_size** | Maximum number of outgoing messages to be queued for delivery to subscribers |
| **connect\_cb** | Function to call when a subscriber connects |
| **disconnect\_cb** | (optional) Function to call when a subscriber disconnects |
| **tracked\_object** | (optional) A shared pointer to an object to track for these callbacks. If set, the a weak\_ptr will be created to this object,  and if the reference count goes to 0 the subscriber callbacks will not get called.  Note that setting this will cause a new reference to be added to the object before the callback, and for it to go out of scope (and potentially be deleted) in the code path (and therefore thread) that the callback is invoked from. |
| **latch** | (optional) If true, the last message published on this topic will be saved and sent to new subscribers when they connect |

**Returns**

On success, a SCNPublisher that, when it goes out of scope, will automatically release a reference on this advertisement. On failure, an empty SCNPublisher.

* 1. Subscribers:

1) template<class M> SCNSubscriber subscribe(const std::string &node\_name,

const std::string &topic\_subscribe,

uint32\_t queue\_size,

void(\*fp)(M),

const TransportHints& transport\_hints = TransportHints())

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **topic** | Topic to subscribe to |
| **queue\_size** | Number of incoming messages to queue up for  processing (messages in excess of this queue capacity will be  discarded). |
| **fp** | Member function pointer to call when a message has arrived |
| **obj** | Object to call fp on |
| **transport\_hints** | a TransportHints structure which defines various transport-related options  \return On success, a Subscriber that, when all copies of it go out of scope, will unsubscribe from this topic. |

**Returns**

On success, a SCNSubscriber that, when all copies of it go out of scope, will unsubscribe from this topic.

On failure, an empty SCNSubscriber which can be checked.

2) template<class M> SCNSubscriber subscribe(const std::string &node\_name,

const std::string& topic\_subscribe,

uint32\_t queue\_size,

void(\*fp)(const boost::shared\_ptr<M const>&),

const TransportHints& transport\_hints = TransportHints())

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **topic** | Topic to subscribe to |
| **queue\_size** | Number of incoming messages to queue up for  processing (messages in excess of this queue capacity will be  discarded). |
| **fp** | Member function pointer to call when a message has arrived |
| **obj** | Object to call fp on |
| **transport\_hints** | a TransportHints structure which defines various transport-related options  \return On success, a Subscriber that, when all copies of it go out of scope, will unsubscribe from this topic. |

**Returns**

On success, a SCNSubscriber that, when all copies of it go out of scope, will unsubscribe from this topic.

On failure, an empty SCNSubscriber which can be checked.

3) template<class M> SCNSubscriber subscribe(const std::string &node\_name,

const std::string& topic\_subscribe,

uint32\_t queue\_size,

const boost::function<void (const boost::shared\_ptr<M const>&)>& callback,

const VoidConstPtr& tracked\_object = VoidConstPtr(),

const TransportHints& transport\_hints = TransportHints())

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **topic** | Topic to subscribe to |
| **queue\_size** | Number of incoming messages to queue up for  processing (messages in excess of this queue capacity will be  discarded). |
| **fp** | Member function pointer to call when a message has arrived |
| **obj** | Object to call fp on |
| **transport\_hints** | a TransportHints structure which defines various transport-related options  \return On success, a Subscriber that, when all copies of it go out of scope, will unsubscribe from this topic. |

**Returns**

On success, a SCNSubscriber that, when all copies of it go out of scope, will unsubscribe from this topic.

On failure, an empty SCNSubscriber which can be checked.

* 1. Servers:

1) template<class T, class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

bool(T::\*srv\_func)(MReq &, MRes &), T \*obj)

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **services\_provided** | Service name to advertise on |
| **srv\_func** | Member function pointer to call when a message has arrived |
| **obj** | Object to call srv\_func on |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

2) template<class T, class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

bool(T::\*srv\_func)(ros::ServiceEvent<MReq, MRes>&), T \*obj)

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **services\_provided** | Service name to advertise on |
| **srv\_func** | Member function pointer to call when a message has arrived |
| **obj** | Object to call srv\_func on |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

3) template<class T, class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

bool(T::\*srv\_func)(MReq &, MRes &),

const boost::shared\_ptr<T>& obj)

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **services\_provided** | Service name to advertise on |
| **srv\_func** | Member function pointer to call when a message has arrived |
| **obj** | Object to call srv\_func on |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

4) template<class T, class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

bool(T::\*srv\_func)(ros::ServiceEvent<MReq, MRes>&), T \*obj)

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **services\_provided** | Service name to advertise on |
| **srv\_func** | Member function pointer to call when a message has arrived |
| **obj** | Object to call srv\_func on |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

5) template<class T, class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

bool(T::\*srv\_func)(ros::ServiceEvent<MReq, MRes>&),

const boost::shared\_ptr<T>& obj)

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **services\_provided** | Service name to advertise on |
| **srv\_func** | Member function pointer to call when a message has arrived |
| **obj** | Object to call srv\_func on |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

6) template<class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided, bool(\*srv\_func)(MReq&, MRes&))

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **services\_provided** | Service name to advertise on |
| **srv\_func** | Member function pointer to call when a message has arrived |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

7) template<class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

bool(\*srv\_func)(ros::ServiceEvent<MReq, MRes>&))

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **services\_provided** | Service name to advertise on |
| **srv\_func** | Member function pointer to call when a message has arrived |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

8) template<class MReq, class MRes>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

const boost::function<bool(MReq&, MRes&)>& callback,

const ros::VoidConstPtr& tracked\_object = VoidConstPtr())

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **service** | Service name to advertise on |
| **callback** | Callback to call when the service is called |
| **tracked\_object** | A shared pointer to an object to track for these callbacks. If set, the a weak\_ptr will be created to this object, and if the reference count goes to 0 the subscriber callbacks will not get called. Note that setting this will cause a new reference to be added to the object before the callback, and for it to go out of scope (and potentially be deleted) in the code path (and therefore thread) that the callback is invoked from. |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

9) template<class S>

ros::SCNServiceServer advertiseService(const std::string &node\_name,

const std::string &services\_provided,

const boost::function<bool(S&)>& callback,

const ros::VoidConstPtr& tracked\_object = VoidConstPtr())

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **service** | Service name to advertise on |
| **callback** | Callback to call when the service is called |
| **tracked\_object** | A shared pointer to an object to track for these callbacks. If set, the a weak\_ptr will be created to this object, and if the reference count goes to 0 the subscriber callbacks will not get called. Note that setting this will cause a new reference to be added to the object before the callback, and for it to go out of scope (and potentially be deleted) in the code path (and therefore thread) that the callback is invoked from. |

**Returns**

On success, a SCNServiceServer that, when all copies of it go out of scope, will unadvertise this service.

On failure, an empty SCNServiceServer which can be checked.

* 1. Clients:

1) template<class MReq, class MRes>

ros::SCNServiceClient serviceClient(const std::string &node\_name,

const std::string &services\_used,

bool persistent = false, const ros::M\_string& header\_values = M\_string())

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **service\_name** | The name of the service to connect to |
| **persistent** | Whether this connection should persist. Persistent services keep the connection to the remote host active so that subsequent calls will happen faster. In general persistent services are discouraged, as they are not as robust to node failure as non-persistent services. |
| **header\_values** | Key/value pairs you'd like to send along in the connection handshake |

2) template<class Service>

ros::SCNServiceClient serviceClient(const std::string &node\_name,

const std::string &services\_used,

bool persistent = false, const ros::M\_string& header\_values = M\_string())

**Parameters**

|  |  |
| --- | --- |
| **node\_name** | Name of the current node |
| **service\_name** | The name of the service to connect to |
| **persistent** | Whether this connection should persist. Persistent services keep the connection to the remote host active so that subsequent calls will happen faster. In general persistent services are discouraged, as they are not as robust to node failure as non-persistent services. |
| **header\_values** | Key/value pairs you'd like to send along in the connection handshake |

3) template<class MReq, class MRes> bool call(SCNNodeHandle &nh, MReq &req, MReq &res)

**Parameters**

|  |  |  |
| --- | --- | --- |
|  | **nh** | The node handler in the current node. |
|  | **req** | The request message. |
| [out] | **res** | Storage for the response message. |

**Returns**

true on success, false otherwise.

4) template<class Service> bool call(SCNNodeHandle &nh, Service& service)

**Parameters**

|  |  |
| --- | --- |
| **nh** | The node handler in the current node. |
| **service** | The service class that contains the request and response messages |

**Returns**

true on success, false otherwise.

6) template<typename MReq, typename MRes> bool call(SCNNodeHandle &nh, const MReq& req, MRes& resp, const std::string& service\_md5sum)

**Parameters**

|  |  |  |
| --- | --- | --- |
|  | **nh** | The node handler in the current node. |
|  | **req** | The request message. |
| [out] | **res** | Storage for the response message. |
|  | **service\_md5sum** | The hash string |

**Returns**

true on success, false otherwise.

# **SCN Services**

Every application node shall provide the callback function to the SCN during the node initialization period through the framework API (scnInit) – One is used to enter/exit reconfiguration mode, and the other one is used to save state of the node.

In the reconfigure mode callback, node developer can specify the following two kinds of services:

**1) Reconfiguration service:**

This service shall be provided by the application node and called by the system control node when the system needs to enter reconfiguration mode. The application developer needs to ensure that all the implementation of the service does everything required to ensure the given node is in reconfiguration mode, as no communication between nodes can be done once a node enters reconfiguration mode. The application developer also needs to keep in mind that it may be possible that the node is killed after it enters reconfiguration mode, so necessary actions to handle this need to be taken when this service is called.

In the save state callback, node developer can specify the following services:

**2) Save state of the node:**

Basically, this is the interface that allows node developer to specify the detailed behaviors of a node to store some necessary information when it is about to enter reconfiguration mode, in case the node is going to be killed.

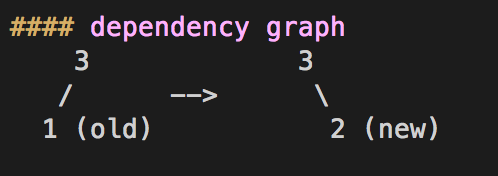
**3) Load state of the node:**

This interface allows the node developer to specify the detailed behavior of a node to load some necessary information when it is starting up during the reconfiguration phase.

# **Detailed Sample Use Case**

In this part, we provided four detailed use cases to demo how to use our framework with different dependency of those nodes. All the related source code are within the folder framework/src/reconfigure/usecase/.

**1) Sample Use Case 1:** this is a basic use case shows how to use the framework with three basic nodes. The only dependency within those nodes are service, and the dependency tree graph can be shown as follows:

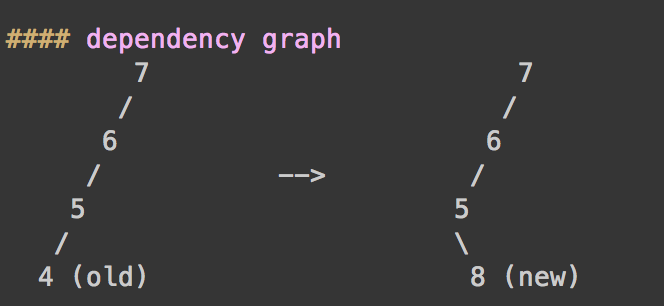


Detailed dependency: demoNode1 provides demoNode1Service, and demoNode2 that will be replaced in this case, provides the same service as demoNode1. And demoNode3 will use demoNode1TestService that provides by demoNode1 and demoNode2. More details and code can be found under framework/src/reconfigure/usecase/1/.

Users can trigger and test this use case by calling:



**2) Sample Use Case 2**: this is a basic use case shows how to use the framework with five basic nodes. The only dependency with those nodes are still service, and the dependency tree graph can be shown as follows:

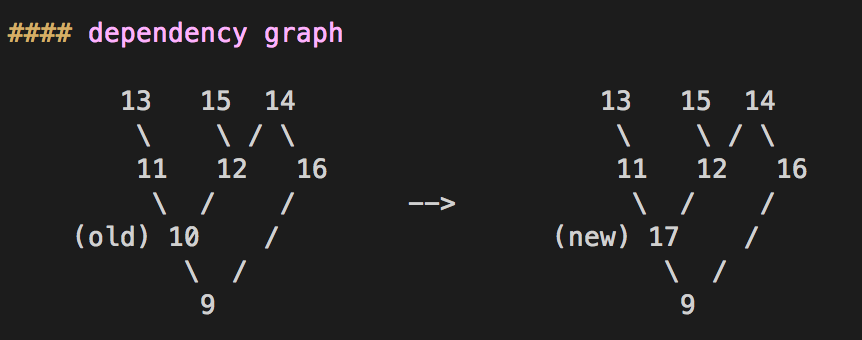


Detailed dependency: it’s clear that the dependency of this use case is linear. demoNode 4 provides demoNode4TestService that demoNode5 uses, and demoNode5 provides demoNode5TestService that demoNode6 uses, and demoNode6 provides demoNode6TestService that demoNode7 uses. Here, we’ll replace demoNode4 with demoNode8 that provides the same service to demoNode4. And the order of putting into dependency tree is demoNode7, demoNode6, demoNode5 and demoNode4. More details can be found under framework/src/reconfigure/usecase/2.

Users can trigger and test this use case by calling:



**3) Sample Use Case 3**: this is a basic use case shows how to use the framework with nine basic nodes. The only dependency with those nodes are still service, and the dependency tree graph can be shown as follows:

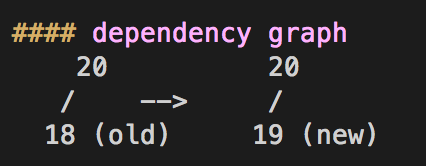


Detailed dependency: the dependency of this use case is more complex compared with the previous two. DemoNode9 provides the demoNode9TestService that’s used by demoNode10 and demoNode16, demoNode10 provides the service demoNode10TestService that demoNode11 and demoNode12 use. Similarly, we can find all the dependency within the graph above. The requirement of this use case is to replace demoNode10 with demoNode17. More details can be found under framework/src/reconfigure/usecase/3.

Users can trigger and test this use case by calling:



**4) Sample Use Case 4:** this is a basic use case shows how to use the framework with three basic nodes. The only dependency is all about topic and the dependency tree graph can be shown as follows:



Detailed dependency: demoNode18 and 19 provide the same service demoNodeTestTopic18 for demoNode20. More details, please refer the framework/src/reconfigure/usecase/4.

Users can trigger and test this use case by calling:



# **Guidelines of Wrapping**

As you can see from the above user guide, our framework has already provided wrappers for basic ROS concepts like NodeHandler, Init, Topic and Service. But whether you want to put the node into reconfiguration mode relies on the node developer, some node may not but put into reconfiguration. Here, I list some guideline of using our framework wrappers.

1. Node is the basic unit of putting into reconfiguration mode, even if only one topic or service that the node provides, uses, publishes, or subscribes. If the dependency tree relies on this node, this node will be put into reconfiguration mode in the order determined by the dependency tree.

2. The order of putting into reconfiguration mode will be determined by the dependency tree the node developer helps to generate. The dependency direction works as follows:

2.1 if node A publishes a topic and subscribed by node B, node B depends on node A.

2.2 if node A provides a service and used by node B, node B depends on node A.

2.3 if node A publishes a topic and subscribed by node B, node B also provides a service and used by node A, then we’ll say node A depends on node B, since we think service has a stronger relationship than topic.

3. If the node developer doesn’t want to use our framework for one node, please abandon our framework totally. That means you cannot use only partial features of our framework.