ROS Crash-Course, Part II

ROS Design Patterns, C++ APIs, and Best Practices

Jonathan Bohren

With some information and figures adapted from `http://www.ros.org`

LABORATORY FOR
Computational
Sensing + Robotics
THE JOHNS HOPKINS UNIVERSITY

Outline

1. ROS Package & Stack Design

Outline

1. ROS Package & Stack Design

2. Integrating ROS With Other Systems

Outline

1. ROS Package & Stack Design
2. Integrating ROS With Other Systems
3. Communication Design Patterns

Outline

1. ROS Package & Stack Design

2. Integrating ROS With Other Systems

3. Communication Design Patterns
    - Datagram Design

Outline

1. ROS Package & Stack Design
2. Integrating ROS With Other Systems
3. Communication Design Patterns
    - Datagram Design
4. ROS (C++) APIs

Outline (revisted)

1. ROS Package & Stack Design
2. Integrating ROS With Other Systems
3. Communication Design Patterns
   - Datagram Design
4. ROS (C++) APIs

Reasons to Factor Code

Splitting code into more packages

- Modularity / Re-usability
  Often, code can be useful in contexts other than those for which it was built. Without splitting code the right way, the dependency graph will have cycles and be invalid.

Reasons to Factor Code

Splitting code into more packages

- Modularity / Re-usability
  Often, code can be useful in contexts other than those for which it was built. Without splitting code the right way, the dependency graph will have cycles and be invalid.
- Dependency Minimization
  It is best to separate out the smallest unit of code that might be used as a dependency. This pattern is often used for separating interface definitions like *.msg* and *.srv* files into their own packages.

Reasons to Factor Code

Splitting code into more packages

- Modularity / Re-usability
  Often, code can be useful in contexts other than those for which it was built. Without splitting code the right way, the dependency graph will have cycles and be invalid.
- Dependency Minimization
  It is best to separate out the smallest unit of code that might be used as a dependency. This pattern is often used for separating interface definitions like *.msg* and *.srv* files into their own packages.
- Wrapper Packages
  ROS's package system can be used as a lightweight way of integrating third-party software that cannot be acquired through a system's package manager.

Reasons to Unify Code

Collecting code into fewer packages

- Application-specific Code
  If a collection of packages are *so* application-specific that they cannot be used separately.

Reasons to Unify Code

Collecting code into fewer packages

- Application-specific Code
  If a collection of packages are *so* application-specific that they cannot be used separately.
- Rapid Development for Experimenting and Prototyping
  Sometimes experimental or "hacked"-together code shouldn't be over-engineered into a number of packages during initial development.

When to Make a Stack

Packaging code for distribution

ROS stacks are best made when there are a collection of packages that, while not necessarily depending on each-other, are useful for a common purpose, act as *companions* to some other package, or make up the components of an application.

When to Make a Stack

Packaging code for distribution

ROS stacks are best made when there are a collection of packages that, while not necessarily depending on each-other, are useful for a common purpose, act as *companions* to some other package, or make up the components of an application.

Some examples of stacks include:

- `laser_pipeline` - packages for grabbing and manipulating laser data

When to Make a Stack

Packaging code for distribution

ROS stacks are best made when there are a collection of packages that, while not necessarily depending on each-other, are useful for a common purpose, act as *companions* to some other package, or make up the components of an application.

Some examples of stacks include:

- `laser_pipeline` - packages for grabbing and manipulating laser data
- `robot_model` - packages for modeling a rigid robot

When to Make a Stack

Packaging code for distribution

ROS stacks are best made when there are a collection of packages that, while not necessarily depending on each-other, are useful for a common purpose, act as *companions* to some other package, or make up the components of an application.

Some examples of stacks include:

- `laser_pipeline` - packages for grabbing and manipulating laser data
- `robot_model` - packages for modeling a rigid robot
- `ros_comm` - packages for the ROS middleware & tools

When to Make a Stack

Packaging code for distribution

ROS stacks are best made when there are a collection of packages that, while not necessarily depending on each-other, are useful for a common purpose, act as *companions* to some other package, or make up the components of an application.

Some examples of stacks include:

- `laser_pipeline` - packages for grabbing and manipulating laser data
- `robot_model` - packages for modeling a rigid robot
- `ros_comm` - packages for the ROS middleware & tools
- `pr2_plugs` - the PR2 autonomous recharge application

Naming Conventions

ROS Style Guidelines

under_scored	CamelCase	camelCase	ALL_CAPS
stack_name	MessageType	funcName()	CONSTANT_NAME
package_name	ServiceType		
file_name	ClassName		
namespace_name			
node_name			
topic_name			
service_name			
liblibrary_name			
variable_name			

Outline (revisted)

1 ROS Package & Stack Design

2 Integrating ROS With Other Systems

3 Communication Design Patterns
    Datagram Design

4 ROS (C++) APIs

Adding ROS Interfaces

Lightweight addition with minimal changes to a codebase

Adding ROS Interfaces

Lightweight addition with minimal changes to a codebase

ALOT OF CODE

"the alot" (c) hyperboleandahalf.blogspot.com

Adding ROS Interfaces

Lightweight addition with minimal changes to a codebase

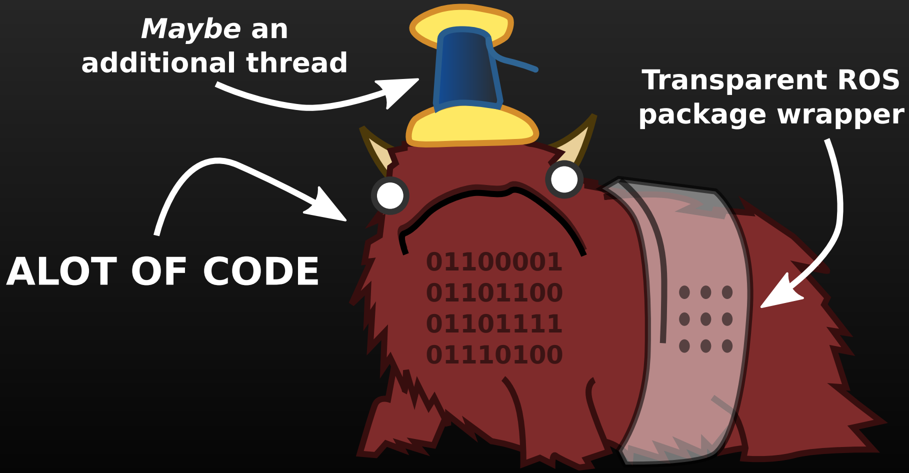
ALOT OF CODE

Transparent ROS package wrapper

"the alot" (c) hyperboleandahalf.blogspot.com

Adding ROS Interfaces

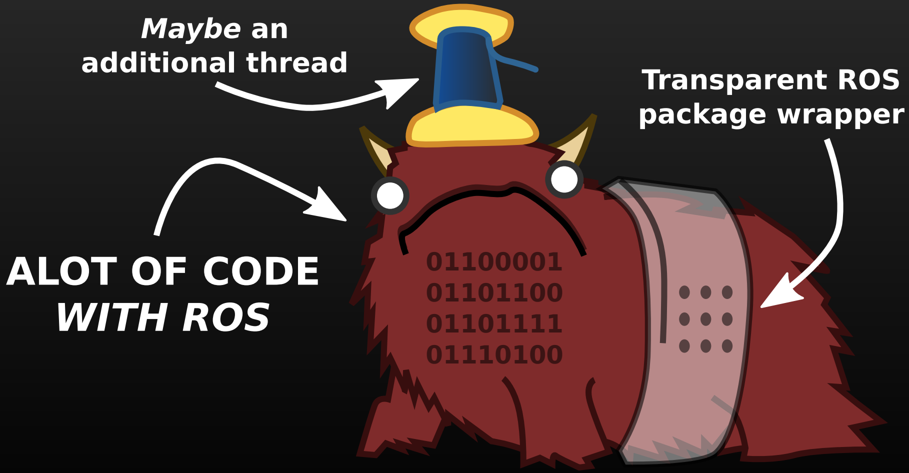
Lightweight addition with minimal changes to a codebase



"the alot" (c) hyperboleandahalf.blogspot.com

Adding ROS Interfaces

Lightweight addition with minimal changes to a codebase



"the alot" (c) hyperboleandahalf.blogspot.com

Outline (revisted)

1. ROS Package & Stack Design
2. Integrating ROS With Other Systems
3. Communication Design Patterns
   - Datagram Design
4. ROS (C++) APIs

.msg Design

ROS .msg design can be summarized in a few basic principles:

- Try to prevent .msg proliferation (ie: try to use existing messages first)

.msg Design

ROS .msg design can be summarized in a few basic principles:

- Try to prevent .msg proliferation (ie: try to use existing messages first)
- Complex messages are built through *composition*

.msg Design

ROS .msg design can be summarized in a few basic principles:

- Try to prevent .msg proliferation (ie: try to use existing messages first)
- Complex messages are built through *composition*
- Message design should come *after* node design

.msg Design

ROS .msg design can be summarized in a few basic principles:

- Try to prevent .msg proliferation (ie: try to use existing messages first)
- Complex messages are built through *composition*
- Message design should come *after* node design
- Try to avoid building messages that tend to not get completely filled out

Namespaces

Like Jersey barriers, but for ROS graph resources

ROS nodes, topics, services, and parameters, can all be created in namespaces, to better organize the collection of names in the ROS graph. Any ROS resoource which is named in a launchfile can be created in a given namespace, using the `<group>` tag and the `ns` attribute.

Namespaces for Designating Topics

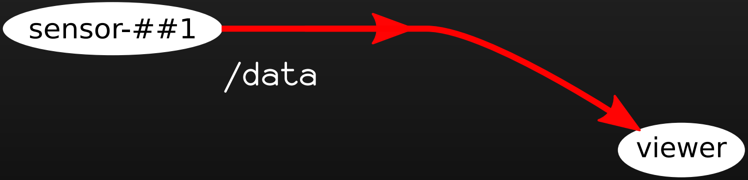
An illustration

sensor-##1

viewer

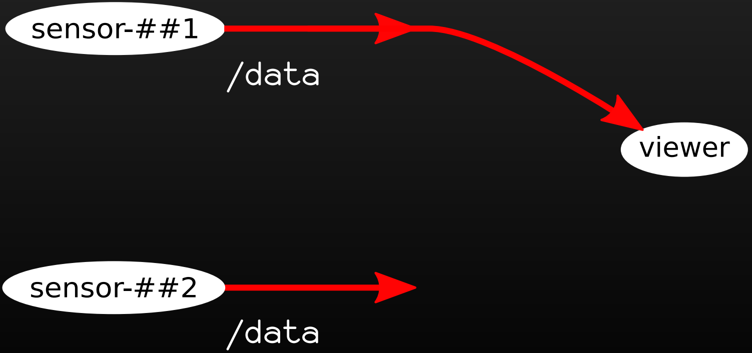
Namespaces for Designating Topics

An illustration



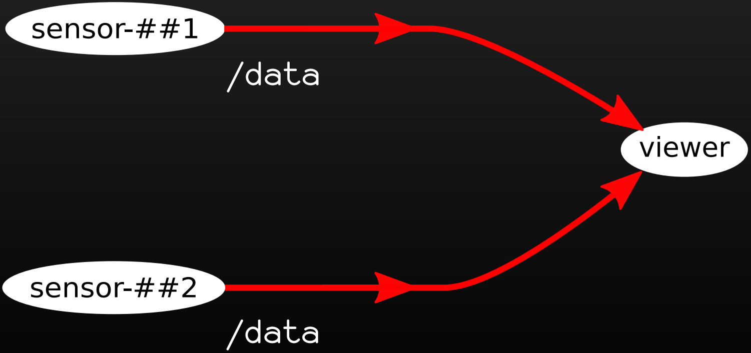
Namespaces for Designating Topics

An illustration



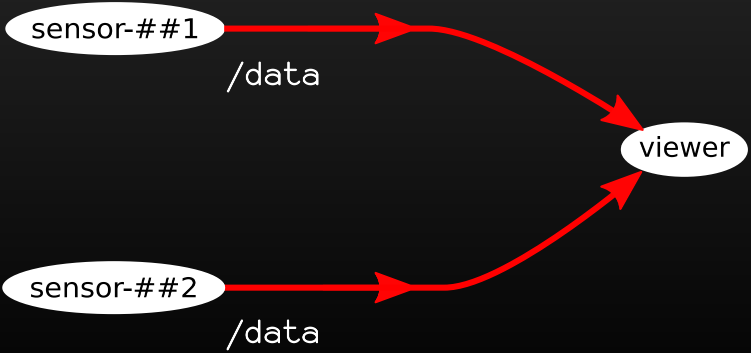
Namespaces for Designating Topics

An illustration



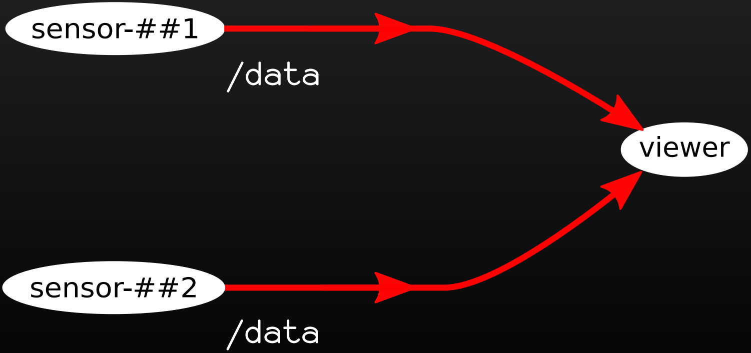
Namespaces for Designating Topics

An illustration



Namespaces for Designating Topics

An illustration



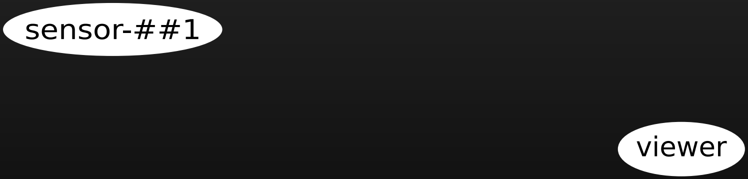
Resource Name “Remapping”

Powerful abstraction interface

One of the most-used and under-documented patterns in ROS communication is resource remapping. While remapping might first appear to simply be another way of reducing naming collisions, it is actually a powerful design tool.

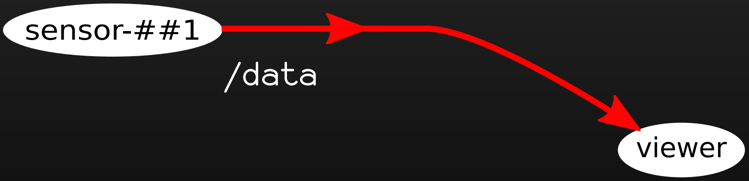
Resource Name “Remapping”

An illustration



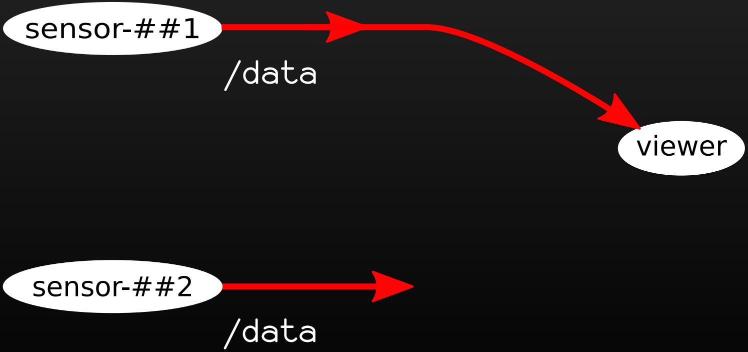
Resource Name “Remapping”

An illustration



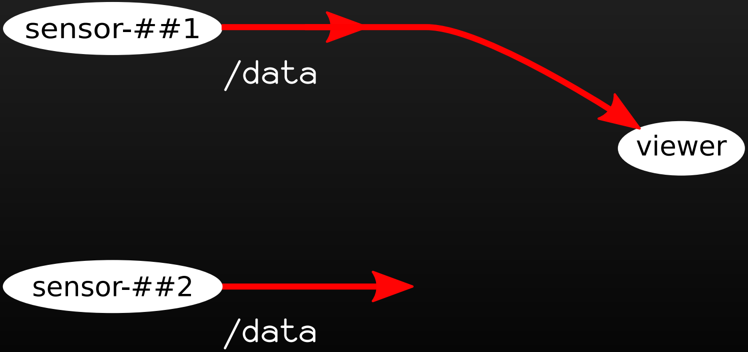
Resource Name “Remapping”

An illustration



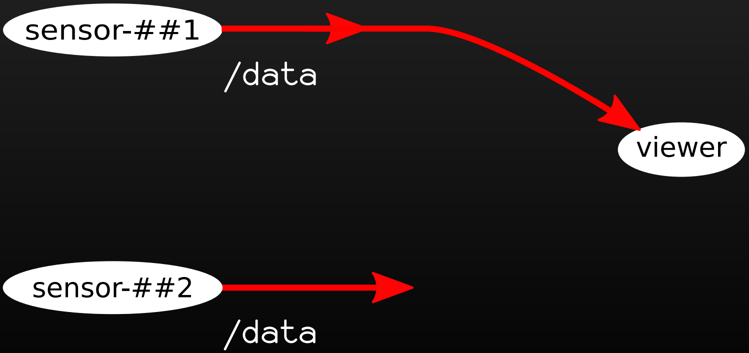
Resource Name "Remapping"

An illustration



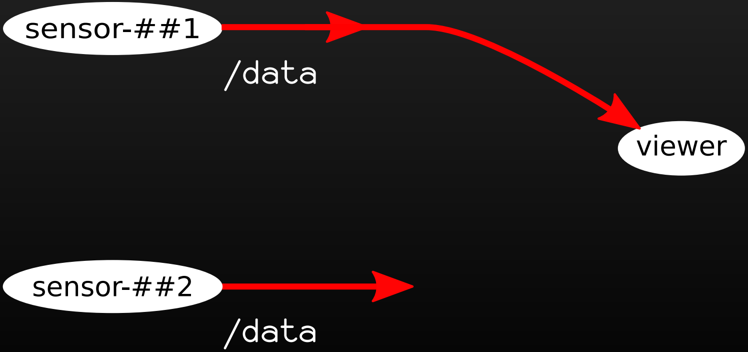
Resource Name “Remapping”

An illustration



Resource Name “Remapping”

An illustration



Outline (revisted)

1. ROS Package & Stack Design
2. Integrating ROS With Other Systems
3. Communication Design Patterns
   - Datagram Design
4. ROS (C++) APIs

Initialization

The first thing you will ever do with ROS

Any progam that uses the ROS APIs must *intialize* the ROS runtime. This is done with a single call to `ros::init()`. This function generally conforms to the format:

```
void ros::init(argc,argc,
               std::string node_name,
               uint32_t options);
```

- `roslaunch` passes arguments into ros via `argc, argv` so these must be passed into your `ros::init()` in order for launchfile settings to take effect.
- Note that if using `roslaunch`, the `node_name` may be over-written
- `options` is a bitfield for advanced use (see online documentation)

“Spinning”

Where all the magic happens

Before, it was claimed that ROS is a *lightweight* architecture, and does not “take over” your program execution. Processing ROS messages and callbacks can be done in a few ways:

- Single-Threaded Programs

“Spinning”

Where all the magic happens

Before, it was claimed that ROS is a *lightweight* architecture, and does not “take over” your program execution. Processing ROS messages and callbacks can be done in a few ways:

- Single-Threaded Programs
  - `ros::spin()`
    Block and process ROS messages and callbacks

“Spinning”

Where all the magic happens

Before, it was claimed that ROS is a *lightweight* architecture, and does not “take over” your program execution. Processing ROS messages and callbacks can be done in a few ways:

- Single-Threaded Programs
  - `ros::spin()`
    Block and process ROS messages and callbacks
  - `ros::spinonce()`
    Block and process only *currently waiting* ROS messages and callbacks (for programs that already have a main loop)

“Spinning”

Where all the magic happens

Before, it was claimed that ROS is a *lightweight* architecture, and does not “take over” your program execution. Processing ROS messages and callbacks can be done in a few ways:

- Single-Threaded Programs
  - `ros::spin()`
    Block and process ROS messages and callbacks
  - `ros::spinonce()`
    Block and process only *currently waiting* ROS messages and callbacks (for programs that already have a main loop)
- Multi-Threaded Programs

“Spinning”

Where all the magic happens

Before, it was claimed that ROS is a *lightweight* architecture, and does not “take over” your program execution. Processing ROS messages and callbacks can be done in a few ways:

- Single-Threaded Programs
  - `ros::spin()`
    Block and process ROS messages and callbacks
  - `ros::spinonce()`
    Block and process only *currently waiting* ROS messages and callbacks (for programs that already have a main loop)
- Multi-Threaded Programs
  - `ros::MultiThreadedSpinner`
    Similar to `ros::spinonce()`, but services callbacks in multiple threads

“Spinning”

Where all the magic happens

Before, it was claimed that ROS is a *lightweight* architecture, and does not “take over” your program execution. Processing ROS messages and callbacks can be done in a few ways:

- Single-Threaded Programs
  - `ros::spin()`
    Block and process ROS messages and callbacks
  - `ros::spinonce()`
    Block and process only *currently waiting* ROS messages and callbacks (for programs that already have a main loop)
- Multi-Threaded Programs
  - `ros::MultiThreadedSpinner`
    Similar to `ros::spinonce()`, but services callbacks in multiple threads
  - `ros::AsynchSpinner`
    Non-blocking, multi-threaded service of callbacks (tends to be the most useful for multi-threaded programs)

Shutting Down

The last thing you will ever do with ROS

ROS programs prefer to terminate cleanly. Doing so is supported through the following mechanisms, which can be called at any time:

- `ros::shutdown()`
  Triggers shutdown of all ROS interfaces in *this node*

Shutting Down

The last thing you will ever do with ROS

ROS programs prefer to terminate cleanly. Doing so is supported through the following mechanisms, which can be called at any time:

- `ros::shutdown()`
  Triggers shutdown of all ROS interfaces in *this node*
- `SIGINT (Ctrl-C from CLI)`
  Same effect as `ros::shutdown()`

Shutting Down

The last thing you will ever do with ROS

ROS programs prefer to terminate cleanly. Doing so is supported through the following mechanisms, which can be called at any time:

- `ros::shutdown()`

  Triggers shutdown of all ROS interfaces in *this node*
- `SIGINT (Ctrl-C from CLI)`

  Same effect as `ros::shutdown()`
- `ros::ok()`

  Returns `false` once *this node's* ROS interfaces have completely shut down

Shutting Down

The last thing you will ever do with ROS

ROS programs prefer to terminate cleanly. Doing so is supported through the following mechanisms, which can be called at any time:

- `ros::shutdown()`
  Triggers shutdown of all ROS interfaces in *this node*
- `SIGINT (Ctrl-C from CLI)`
  Same effect as `ros::shutdown()`
- `ros::ok()`
  Returns `false` once *this node's* ROS interfaces have completely shut down
- `ros::isShuttingDown()`
  Discouraged except in advanced cases, returns `true` once `ros::shutdown()` has been called

Time & The ROS Clock

FIXME: 2038 A.D.

ROS provides a time abstraction interface to provide a mechanism to spoof the wall clock for simulation. ROS time is preferred over a system's clock routines (for non-realtime systems). The `ros::Time` API has the following features:

- `ros::Time` - (int32 sec, int32 μsec) time point

Time & The ROS Clock

FIXME: 2038 A.D.

ROS provides a time abstraction interface to provide a mechanism to spoof the wall clock for simulation. ROS time is preferred over a system's clock routines (for non-realtime systems). The `ros::Time` API has the following features:

- `ros::Time` - (int32 sec, int32 μsec) time point
- `ros::Time::now()` - the current time

Time & The ROS Clock

FIXME: 2038 A.D.

ROS provides a time abstraction interface to provide a mechanism to spoof the wall clock for simulation. ROS time is preferred over a system's clock routines (for non-realtime systems). The `ros::Time` API has the following features:

- `ros::Time` - (int32 sec, int32 μsec) time point
- `ros::Time::now()` - the current time
- `ros::Duration` - (int32 sec, int32 μsec) time *duration*

Time & The ROS Clock

FIXME: 2038 A.D.

ROS provides a time abstraction interface to provide a mechanism to spoof the wall clock for simulation. ROS time is preferred over a system's clock routines (for non-realtime systems). The `ros::Time` API has the following features:

- `ros::Time` - (int32 sec, int32 μsec) time point
- `ros::Time::now()` - the current time
- `ros::Duration` - (int32 sec, int32 μsec) time *duration*
- `ros::Duration::sleep()` - sleep for this duration

Time & The ROS Clock

FIXME: 2038 A.D.

ROS provides a time abstraction interface to provide a mechanism to spoof the wall clock for simulation. ROS time is preferred over a system's clock routines (for non-realtime systems). The `ros::Time` API has the following features:

- `ros::Time` - (int32 sec, int32 μsec) time point
- `ros::Time::now()` - the current time
- `ros::Duration` - (int32 sec, int32 μsec) time *duration*
- `ros::Duration::sleep()` - sleep for this duration
- `ros::[Time/Duration]::toSec()` - (double) in seconds

Time & The ROS Clock

FIXME: 2038 A.D.

ROS provides a time abstraction interface to provide a mechanism to spoof the wall clock for simulation. ROS time is preferred over a system's clock routines (for non-realtime systems). The `ros::Time` API has the following features:

- `ros::Time` - (int32 sec, int32 μsec) time point
- `ros::Time::now()` - the current time
- `ros::Duration` - (int32 sec, int32 μsec) time *duration*
- `ros::Duration::sleep()` - sleep for this duration
- `ros::[Time/Duration]::toSec()` - (double) in seconds
- `ros::Rate` - fixed-rate duration for sleeping