# Introduction to ROS and the Scarab Robots

Amanda Prorok
University of Pennsylvania, GRASP Lab.

TerraSwarm - Programming the Swarm Workshop
UC Berkeley, May 28, 2015

# Overview

1. ROS: concepts
   - computation graph
   - topic negotiation
   - network configuration
   - publisher / subscriber example
2. Using ROS with the Scarab robot
   - setup
   - publisher / subscriber example
   - using roslaunch
3. Application:
   - sending waypoints to the Scarab (**movies**)
   - simulator (**demo**)
4. Interfacing ROS with Swarmlets
   - 2 suggestions

# What is ROS?

ROS is an open-source, meta-operating system, see www.ros.org:

*"The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. "*

ROS provides:

- a collection of package management and software building tools
- implementation of commonly-used functionality
- architecture for distributed inter-process and inter-machine communication and configuration
- hardware abstraction & low-level device control

# ROS Filesystem

ROS resources that you encounter on disk:

| | |
|---|---|
| **packages:** | Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files. |
| **repositories:** | A collection of packages which share a common version control system. |
| **package manifests:** | Manifests (package.xml) provide metadata about a package |
| **message types:** | Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS |
| **service types:** | Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS |

Navigating the filesystem:

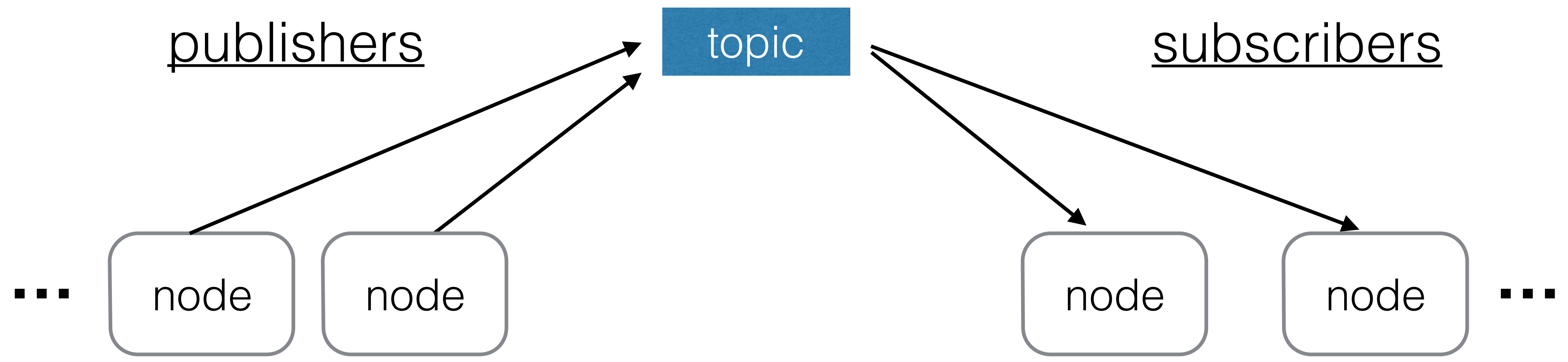
```
roscd, rospack, rosls
```

# ROS Graph Concepts

ROS runtime **graph** is a peer-to-peer network of processes that are loosely coupled via the ROS communication infrastructure

| | |
|---|---|
| **nodes:** | A node is an executable that uses ROS to communicate with other nodes. |
| **messages:** | ROS data type used when subscribing or publishing to a topic. |
| **topics:** | Nodes can publish messages to a topic as well as subscribe to a topic to receive messages. |
| **master:** | Name service for ROS: name registration and lookup; negotiates connections |
| **rosout:** | ROS equivalent of stdout/stderr |
| **parameter server:** | Stores persistent configuration parameters |
| **roscore:** | 3 programs: master + rosout + parameter server |

# ROS: Nodes and Topics



**Nodes:**

- Nodes are processes that perform computation.
- A ROS node is written with the use of a ROS client library: **roscpp** or **rospy**

**Topics:**

- Messages are routed via a transport system with publish / subscribe semantics
- A node sends out a message by publishing it to a given topic
- The topic is a name that is used to identify the content of the message
- Analogy: topic as a strongly typed message bus (according to .msg spec)
- Asynchronous ‘stream-like’ communication
- TCP/IP or UDP transport

# Message Types

ROS uses a simplified Message description language for describing the data values (Messages) that ROS nodes publish. Message descriptions are stored in `.msg` files in the `msg/` subdirectory of a ROS package.

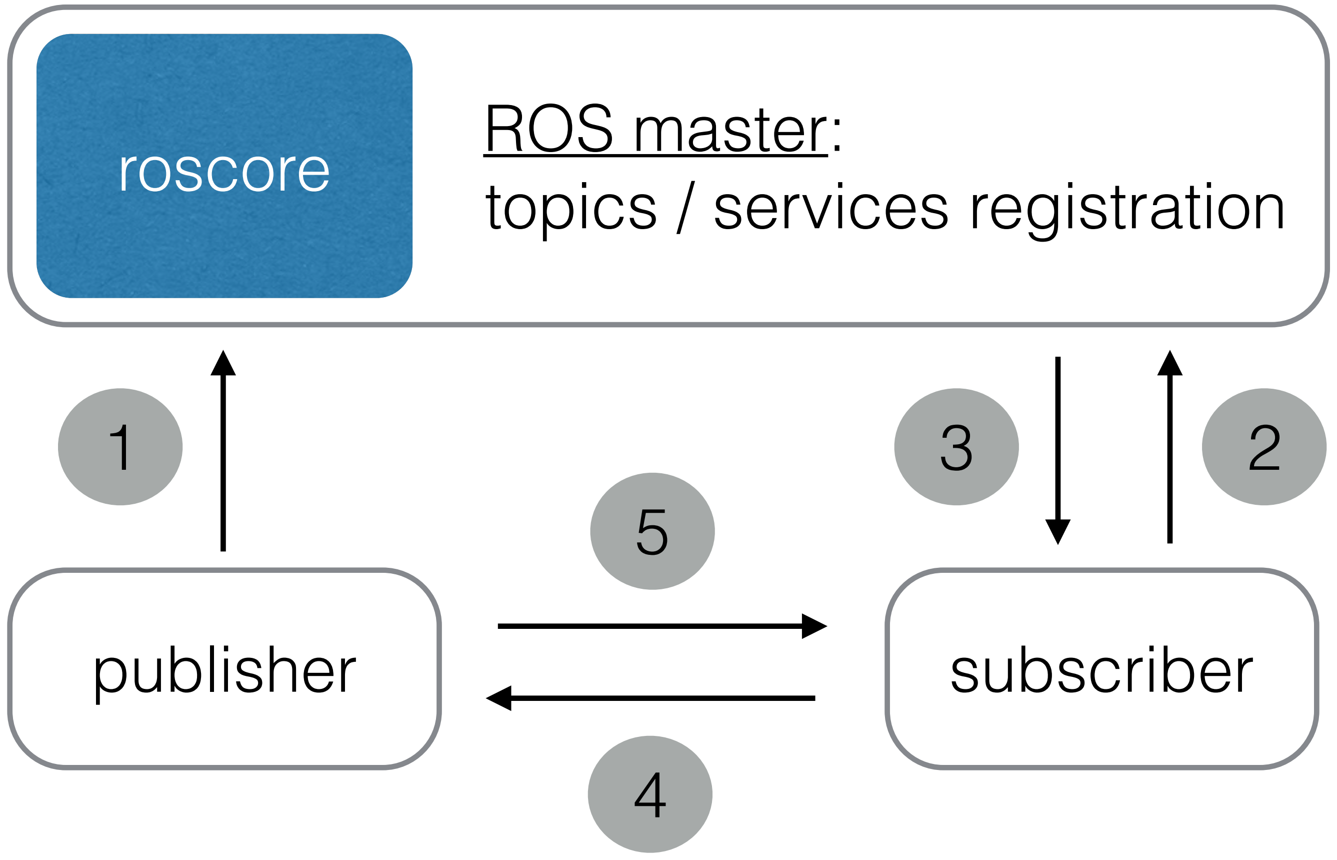
The format of a Message description is a list of data field descriptions and constant definitions on separate lines:

```
int32 x
int32 y
```

Field types can be:

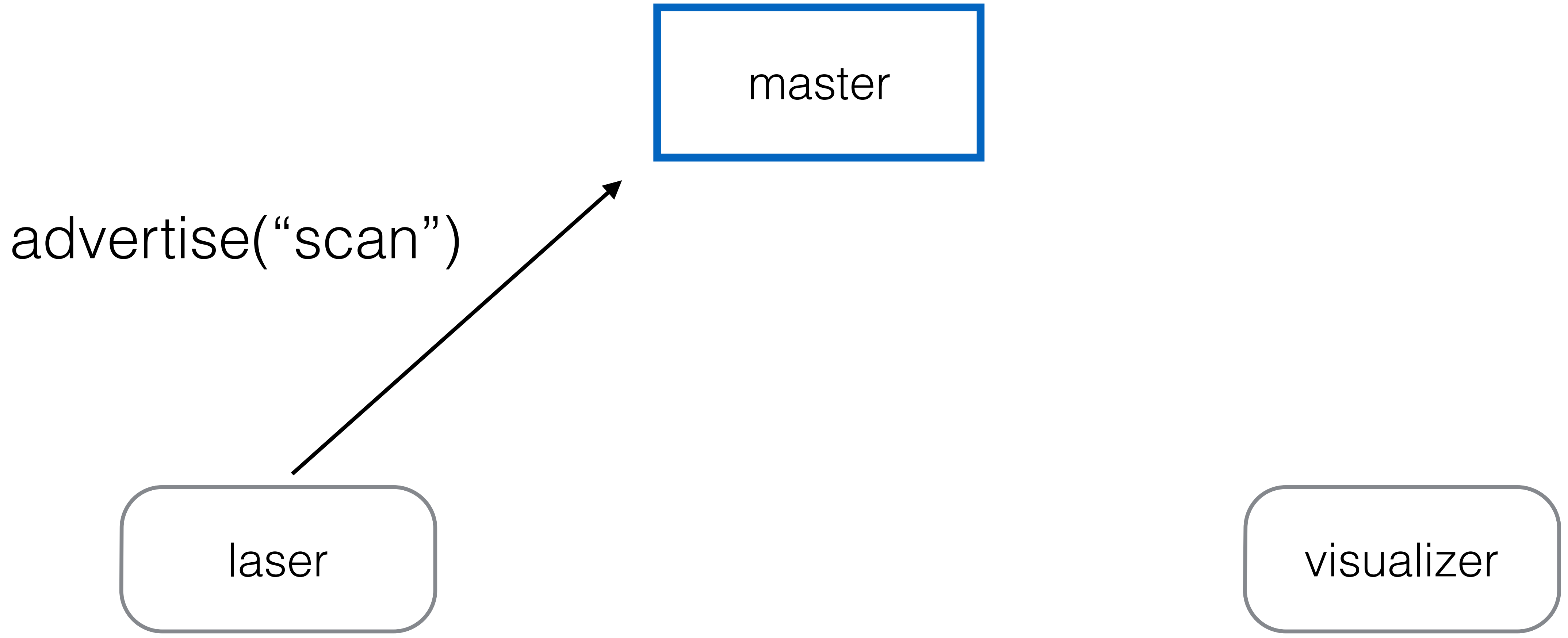
1. a built-in type
2. names of custom or built-in Message descriptions
3. fixed- or variable-length arrays (lists) of the above
4. the special Header type, which maps to std_msgs/Header

# Topic Negotiation

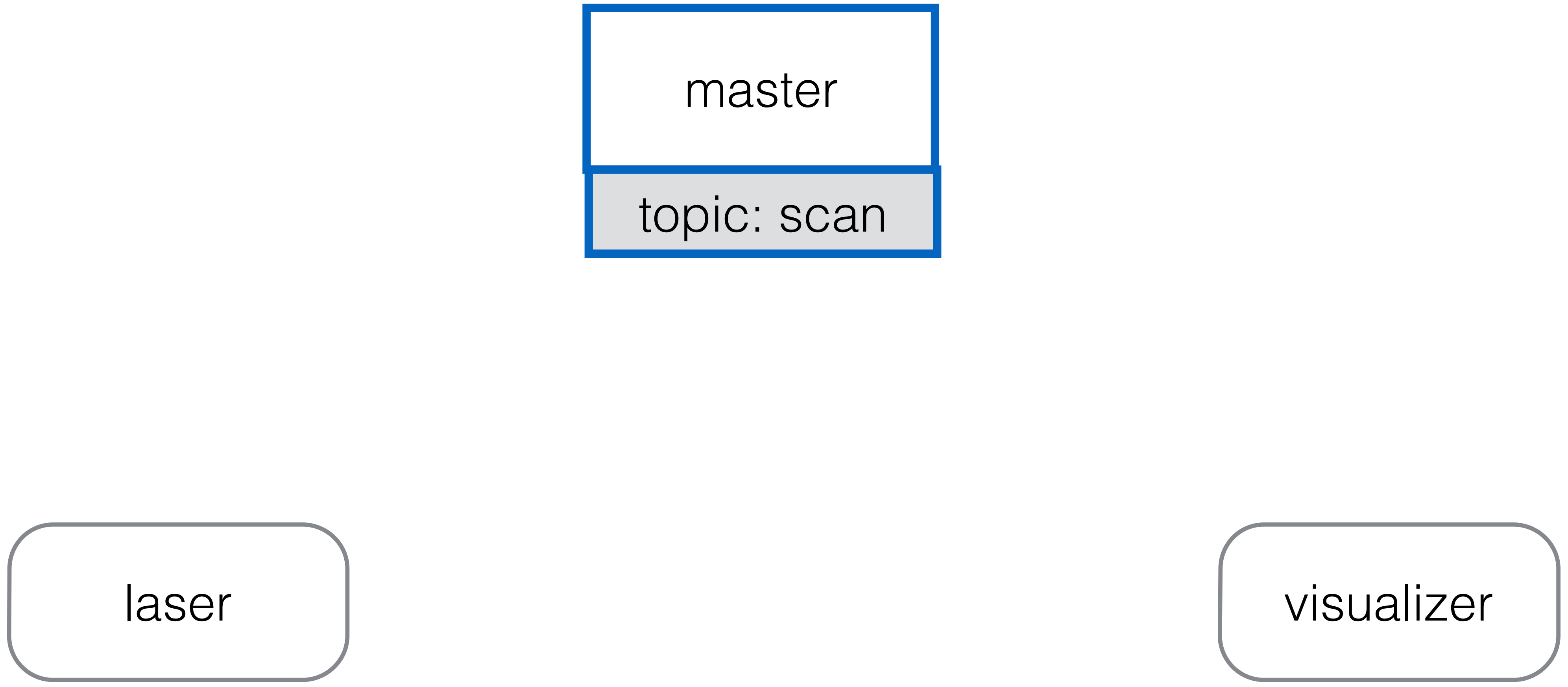


1. I am publishing on /topic, reach me at HOSTNAME
2. I am subscribing to /topic
3. Contact “publisher” at HOSTNAME
4. I am subscribing to /topic
5. Data on /topic

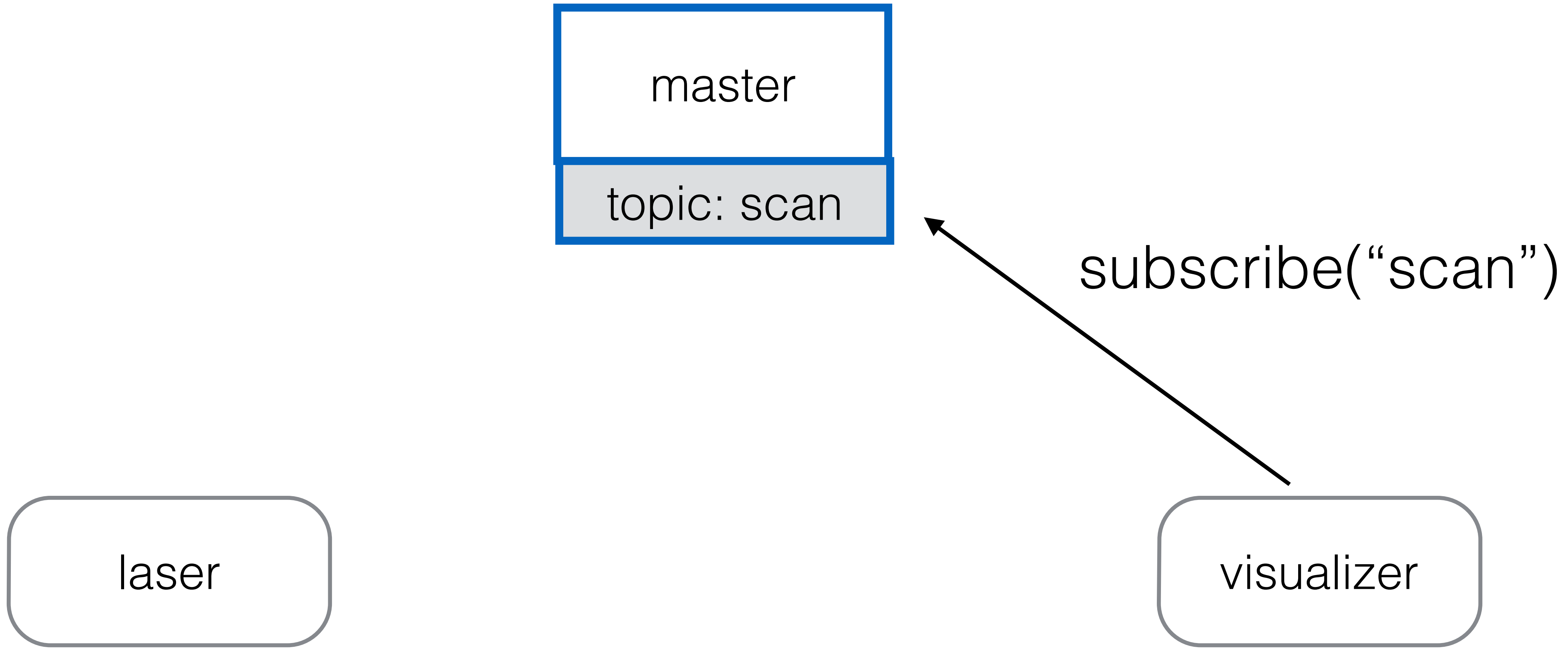
# ROS TCP Topics



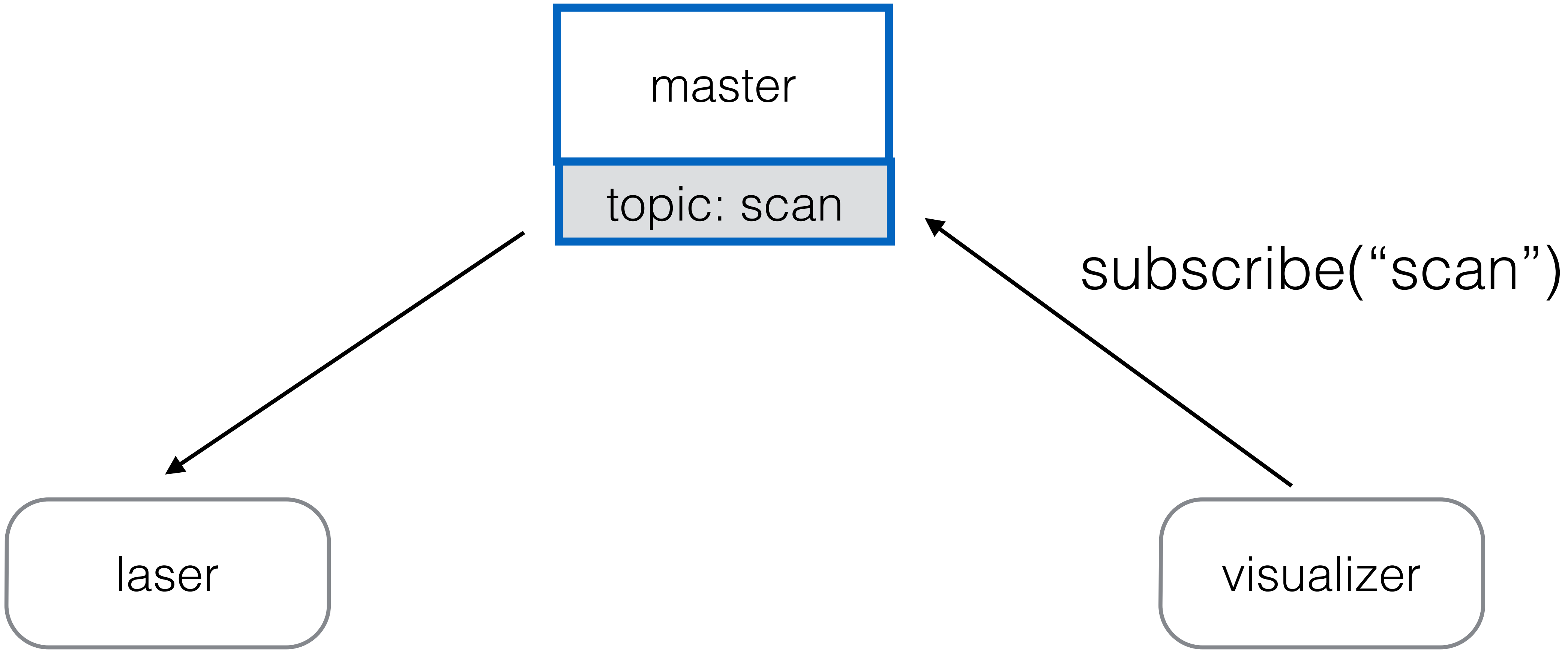
# ROS TCP Topics



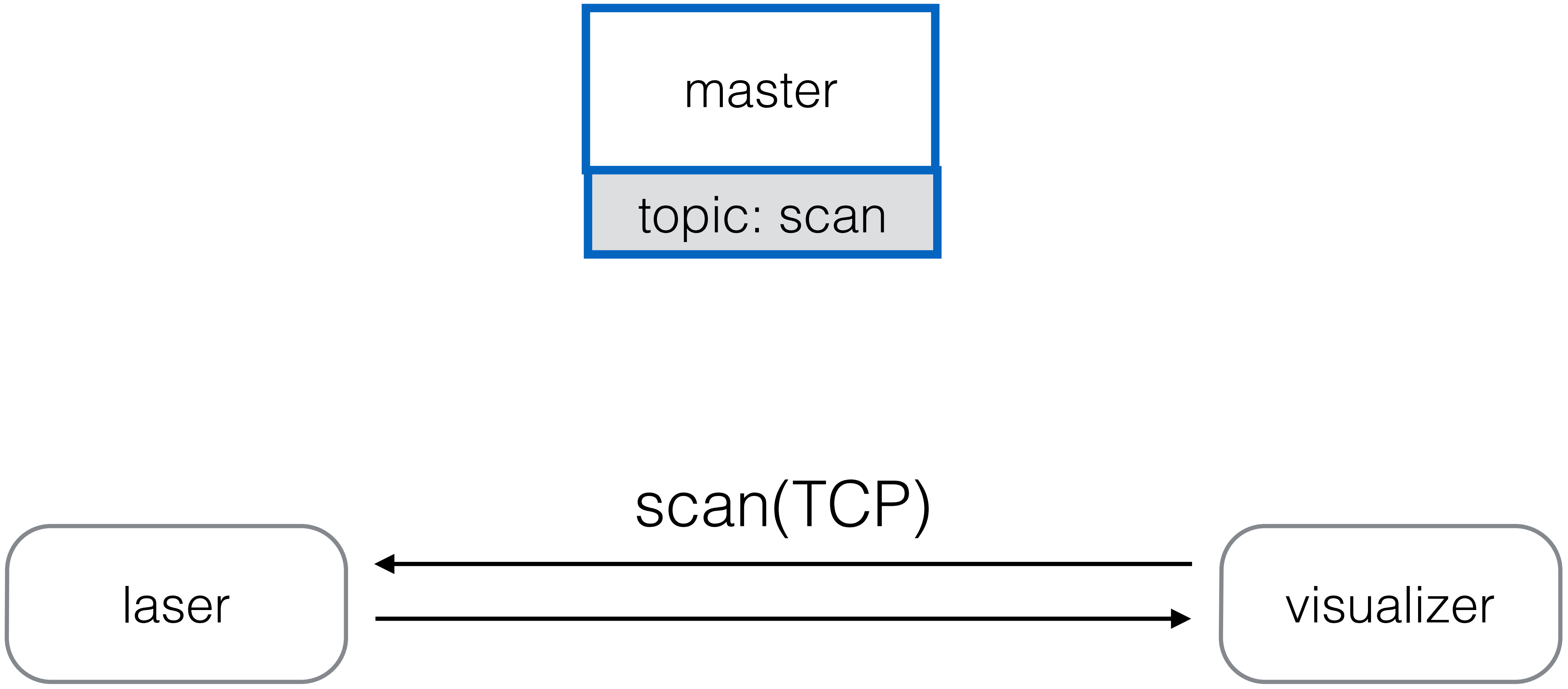
# ROS TCP Topics



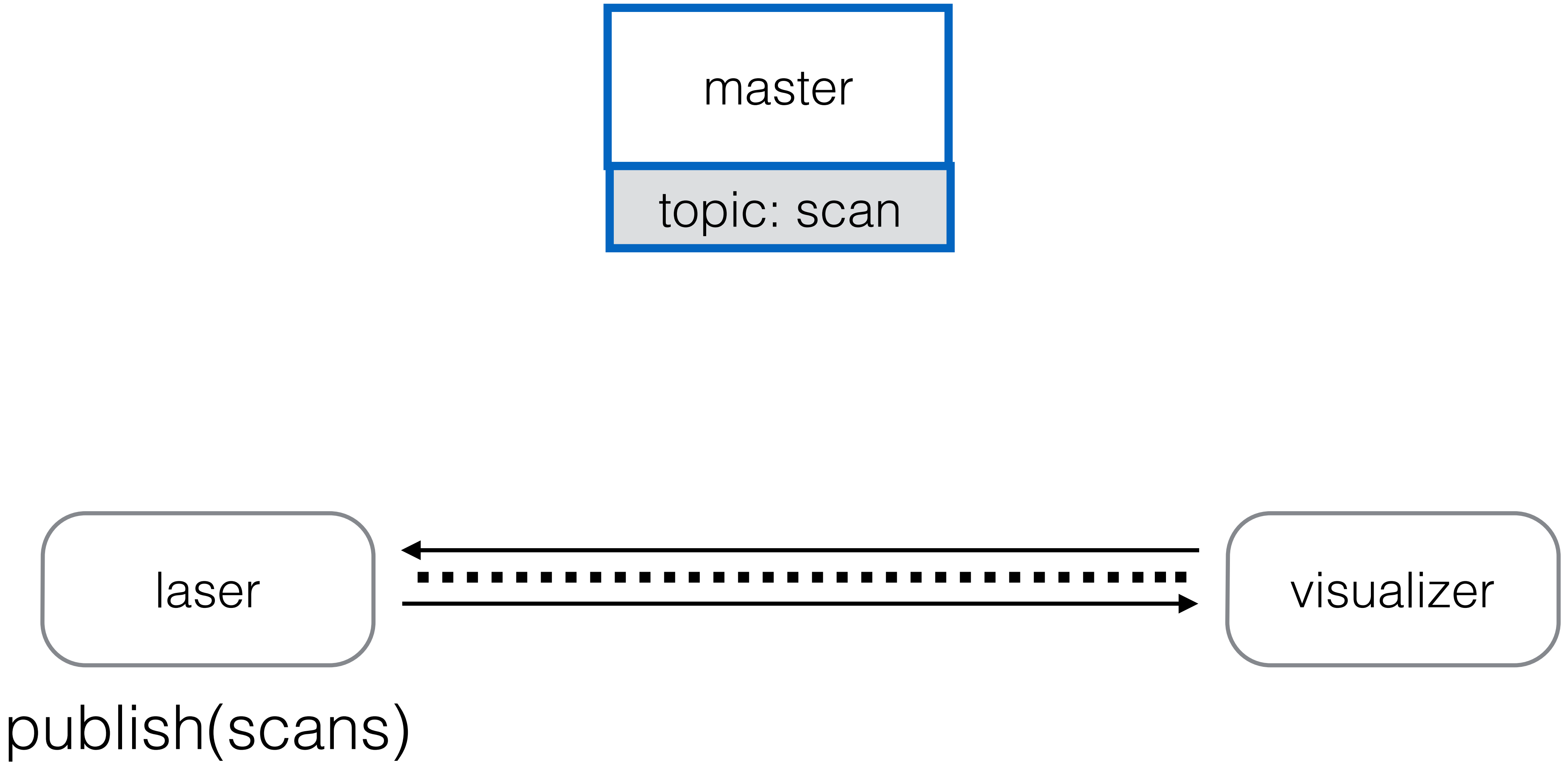
# ROS TCP Topics



# ROS TCP Topics



## ROS TCP Topics



# Network Configuration

`ROS_MASTER_URI` is a required setting that tells nodes where they can locate the master. It should be set to the XML-RPC URI of the master.

When a ROS node advertises a topic, it provides a hostname:port combination (a URI) that other nodes will contact when they want to subscribe to that topic. It is important that the hostname that a node provides can be used by all other nodes to contact it.

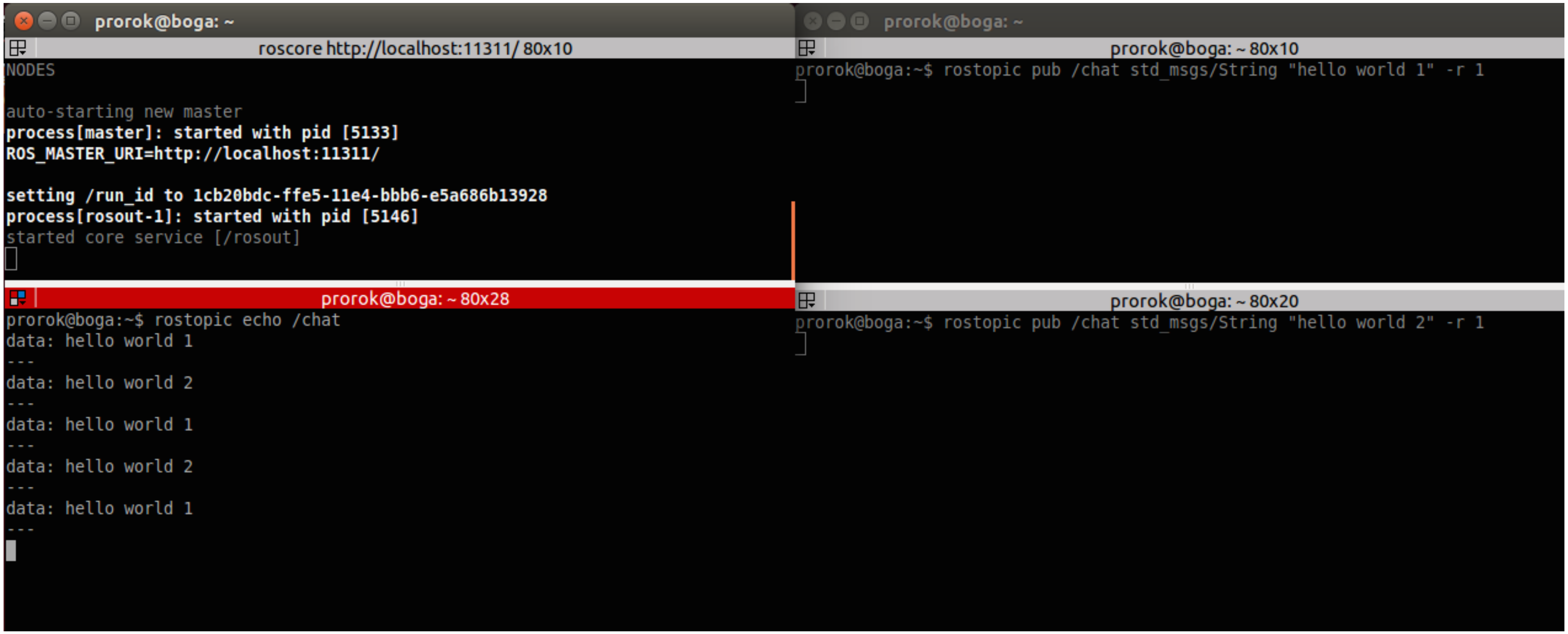
Single machine configuration:

```
$ export ROS_HOSTNAME=localhost
$ export ROS_MASTER_URI=http://localhost:11311
```

Multiple machines configuration:

```
$ export ROS_HOSTNAME=scarab42.wifi
$ export ROS_MASTER_URI=http://scarab44.wifi:11311
```

# Publisher/Subscriber: Example



prorok@boga: ~

roscore http://localhost:11311/ 80x10

```
NODES

auto-starting new master
process[master]: started with pid [5133]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to 1cb20bdc-ffe5-11e4-bbb6-e5a686b13928
process[rosout-1]: started with pid [5146]
started core service [/rosout]
```

prorok@boga: ~ 80x28

```
prorok@boga:~$ rostopic echo /chat
data: hello world 1
---
data: hello world 2
---
data: hello world 1
---
data: hello world 2
---
data: hello world 1
---
```

prorok@boga: ~

prorok@boga: ~ 80x10

```
prorok@boga:~$ rostopic pub /chat std_msgs/String "hello world 1" -r 1
```

prorok@boga: ~ 80x20

```
prorok@boga:~$ rostopic pub /chat std_msgs/String "hello world 2" -r 1
```

# Useful Commands

```
$ rosnode list
$ rosnode info [node_name]
$ rosrun [package_name] [node_name]
$ rostopic echo [topic_name]
$ rostopic type [topic]
$ rostopic pub [topic] [msg_type] [args]
$ rospack find [package_name]
```

Useful links:
http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics
http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
http://wiki.ros.org/ROS/NetworkSetup

## Using ROS with Scarabs

## Setup

1. Install Ubuntu 14.04 (on binary compatible machine)
2. Install the full desktop version of ROS Indigo
3. Configure ROS environment, see instructions:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

4. Install dependencies (see Appendix)
5. Download (and compile) Ben Charrow's *scarab* repository

   https://github.com/bcharrow/scarab/tree/bcharrow-devel

6. Configure your .bashrc file on all machines (including robots):

```
$ export ROS_HOSTNAME=myhostname
$ export ROS_MASTER_URI=http://masterhostname:11311
```

# Scarab Packages

Repository:
https://github.com/bcharrow/scarab/tree/bcharrow-devel

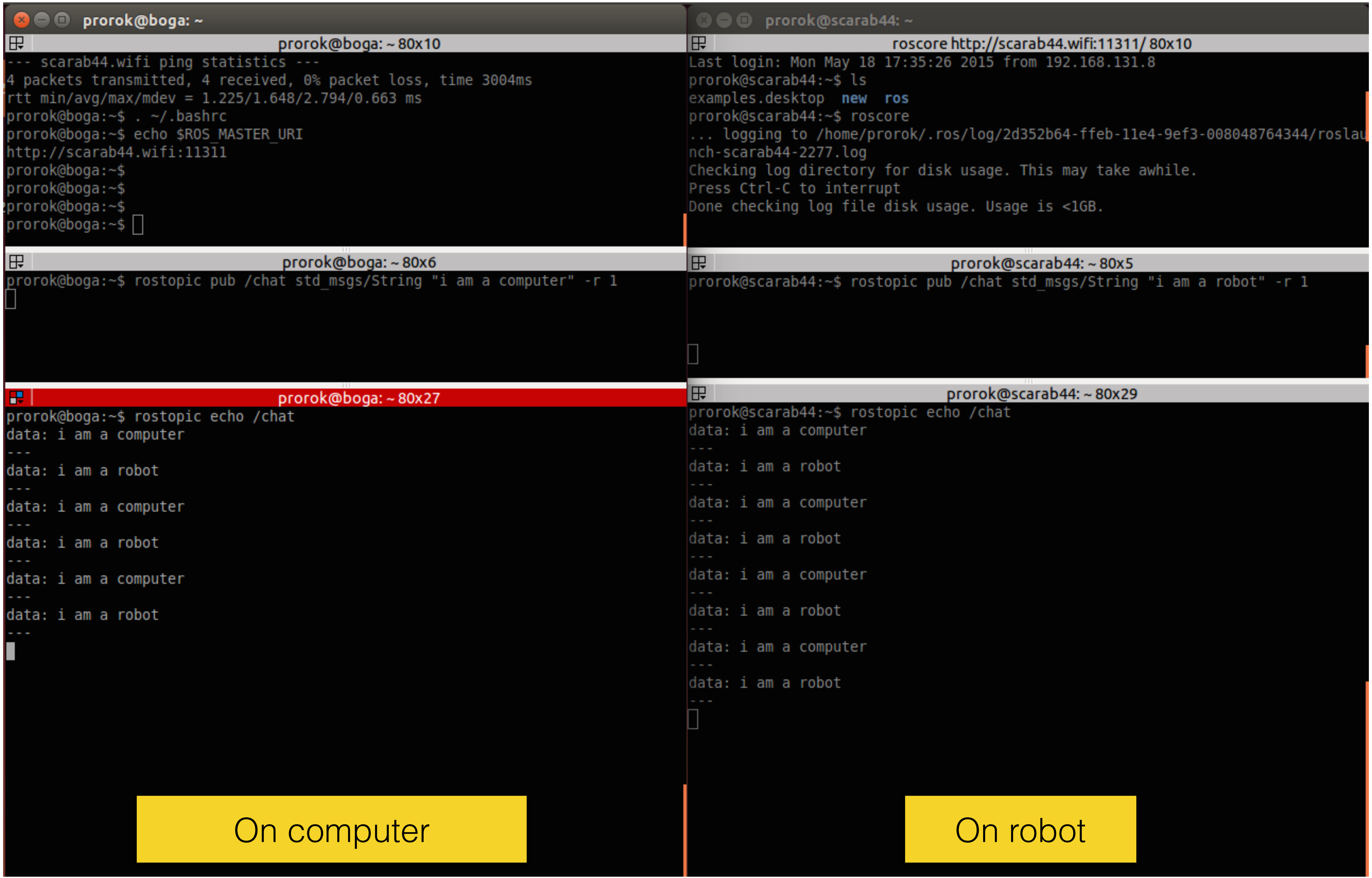
Main packages:

- roboclaw: motor driver
- laser_odom: odometry estimate
- scarab: launch files
- scarab_twist: keyboard driving
- hfn: human friendly navigation

Also used from ROS repository (included in Indigo installation):

- gmapping: occupancy grid mapping

# Publisher / Subscriber: Example



prorok@boga: ~

prorok@boga: ~ 80x10

```
--- scarab44.wifi ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.225/1.648/2.794/0.663 ms
prorok@boga:~$ . ~/.bashrc
prorok@boga:~$ echo $ROS_MASTER_URI
http://scarab44.wifi:11311
prorok@boga:~$
prorok@boga:~$
prorok@boga:~$
prorok@boga:~$
```

prorok@boga: ~ 80x6

```
prorok@boga:~$ rostopic pub /chat std_msgs/String "i am a computer" -r 1
```

prorok@boga: ~ 80x27

```
prorok@boga:~$ rostopic echo /chat
data: i am a computer
---
data: i am a robot
---
data: i am a computer
---
data: i am a robot
---
data: i am a computer
---
data: i am a robot
---
```

On computer

prorok@scarab44: ~

roscore http://scarab44.wifi:11311/ 80x10

```
Last login: Mon May 18 17:35:26 2015 from 192.168.131.8
prorok@scarab44:~$ ls
examples.desktop  new  ros
prorok@scarab44:~$ roscore
... logging to /home/prorok/.ros/log/2d352b64-ffeb-11e4-9ef3-008048764344/roslau
nch-scarab44-2277.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

prorok@scarab44: ~ 80x5

```
prorok@scarab44:~$ rostopic pub /chat std_msgs/String "i am a robot" -r 1
```

prorok@scarab44: ~ 80x29

```
prorok@scarab44:~$ rostopic echo /chat
data: i am a computer
---
data: i am a robot
---
data: i am a computer
---
data: i am a robot
---
data: i am a computer
---
data: i am a robot
---
data: i am a computer
---
data: i am a robot
---
```

On robot

## Launching Multiple Nodes

ROS provides a way to configure and launch a collection of ROS nodes. Use the **`roslaunch`** program with a `.launch` XML file that configures system:

```
roslaunch [packagename] [file.launch]
```

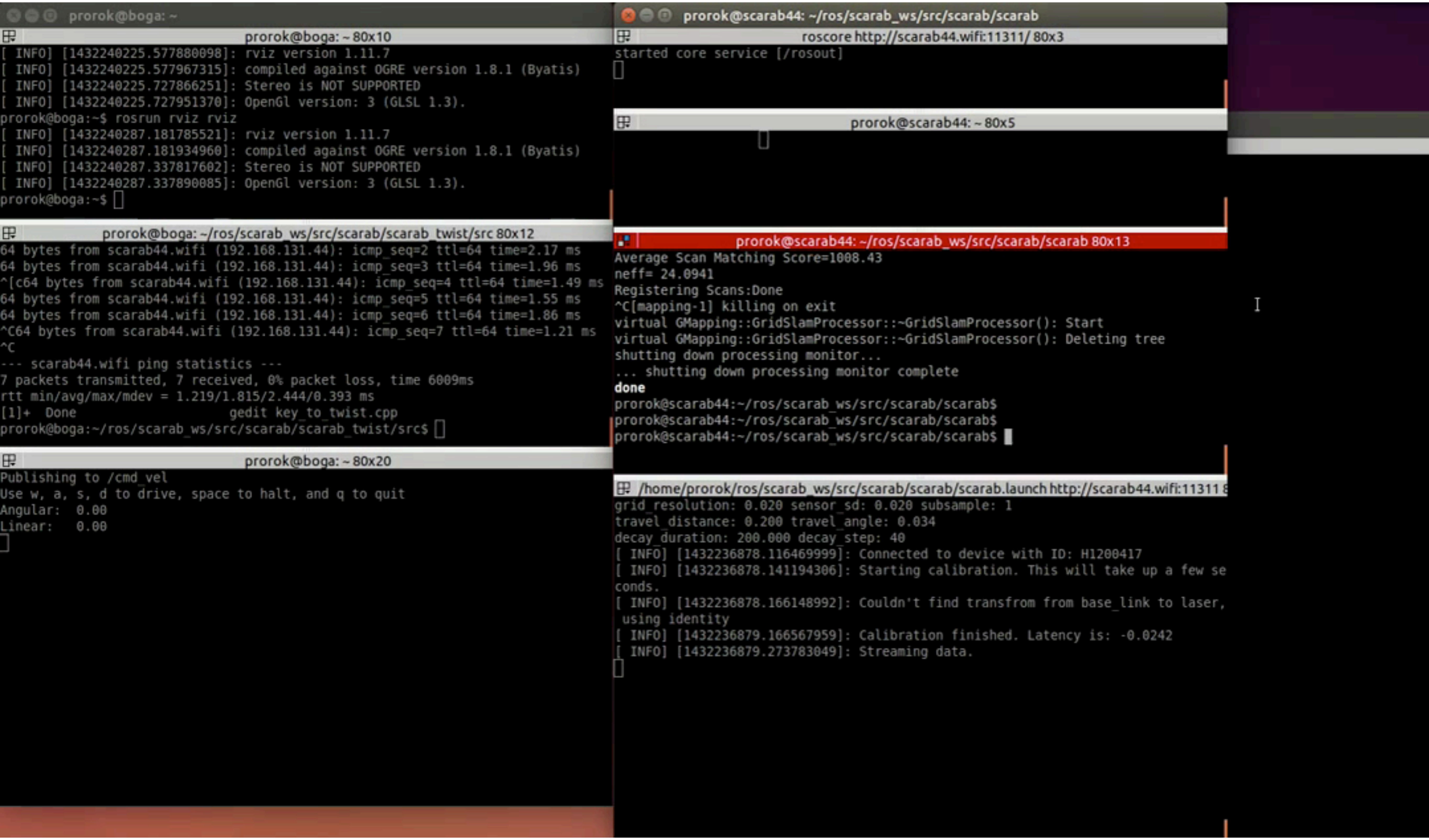
Allows us to:

- easily run multiple ROS nodes locally and remotely via SSH
- set parameters on the parameter server
- automatically re-spawn nodes if they die
- change node names, namespaces, topic names without recompiling

Examples:

```
$ roslaunch scarab scarab.launch
$ roslaunch scarab gmapping.launch
$ roslaunch scarab hfn.launch
```

## Example: Create Map



prorok@boga: ~

prorok@boga: ~ 80x10
```
[ INFO] [1432240225.577880098]: rviz version 1.11.7
[ INFO] [1432240225.577967315]: compiled against OGRE version 1.8.1 (Byatis)
[ INFO] [1432240225.727866251]: Stereo is NOT SUPPORTED
[ INFO] [1432240225.727951370]: OpenGl version: 3 (GLSL 1.3).
prorok@boga:~$ rosrun rviz rviz
[ INFO] [1432240287.181785521]: rviz version 1.11.7
[ INFO] [1432240287.181934960]: compiled against OGRE version 1.8.1 (Byatis)
[ INFO] [1432240287.337817602]: Stereo is NOT SUPPORTED
[ INFO] [1432240287.337890085]: OpenGl version: 3 (GLSL 1.3).
prorok@boga:~$
```

prorok@boga: ~/ros/scarab_ws/src/scarab/scarab_twist/src 80x12
```
64 bytes from scarab44.wifi (192.168.131.44): icmp_seq=2 ttl=64 time=2.17 ms
64 bytes from scarab44.wifi (192.168.131.44): icmp_seq=3 ttl=64 time=1.96 ms
^[c64 bytes from scarab44.wifi (192.168.131.44): icmp_seq=4 ttl=64 time=1.49 ms
64 bytes from scarab44.wifi (192.168.131.44): icmp_seq=5 ttl=64 time=1.55 ms
64 bytes from scarab44.wifi (192.168.131.44): icmp_seq=6 ttl=64 time=1.86 ms
^C64 bytes from scarab44.wifi (192.168.131.44): icmp_seq=7 ttl=64 time=1.21 ms
^C
--- scarab44.wifi ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6009ms
rtt min/avg/max/mdev = 1.219/1.815/2.444/0.393 ms
[1]+  Done                    gedit key_to_twist.cpp
prorok@boga:~/ros/scarab_ws/src/scarab/scarab_twist/src$
```

prorok@boga: ~ 80x20
```
Publishing to /cmd_vel
Use w, a, s, d to drive, space to halt, and q to quit
Angular:   0.00
Linear:    0.00
```

prorok@scarab44: ~/ros/scarab_ws/src/scarab/scarab

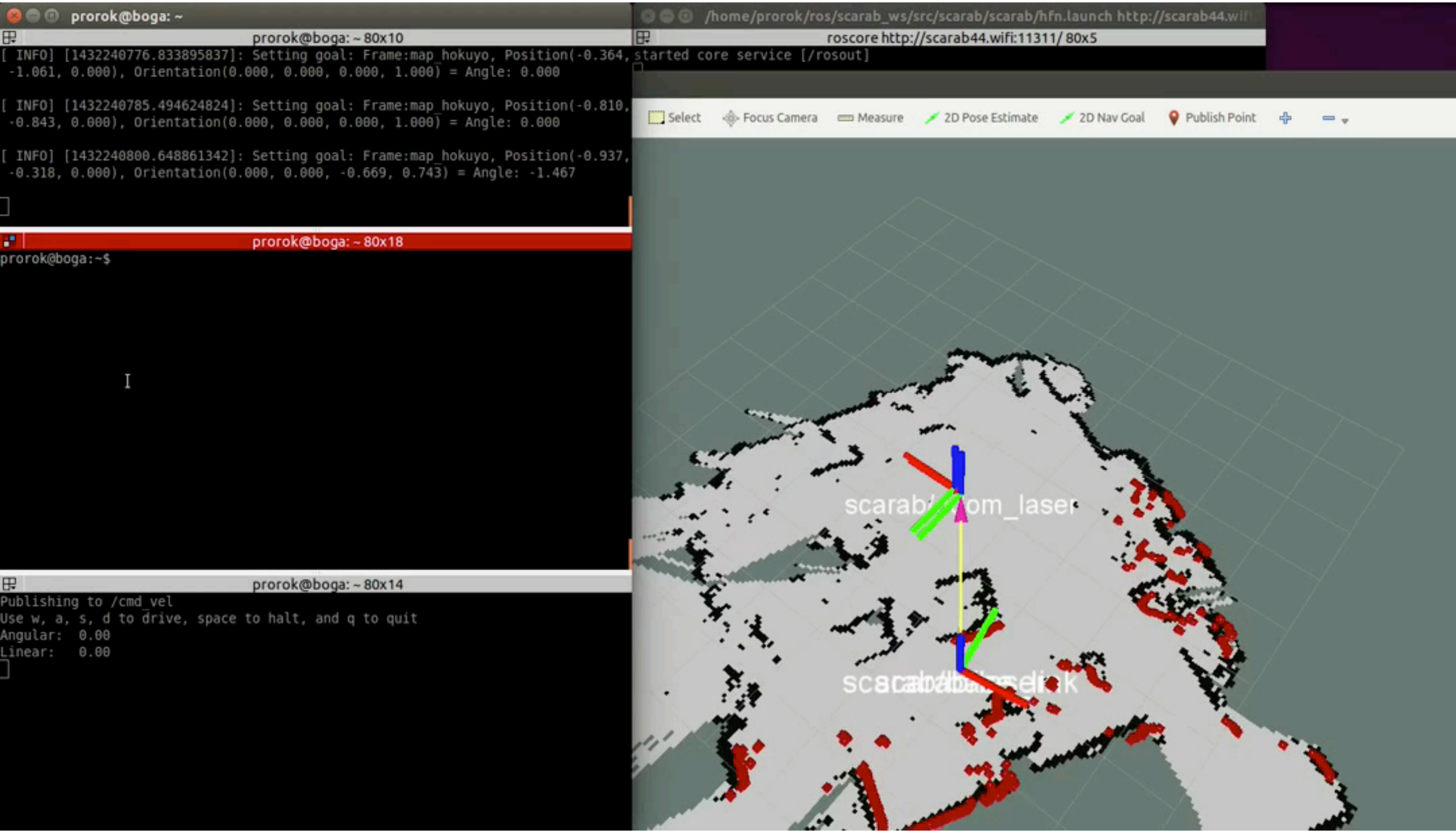
roscore http://scarab44.wifi:11311/ 80x3
```
started core service [/rosout]
```

prorok@scarab44: ~ 80x5

prorok@scarab44: ~/ros/scarab_ws/src/scarab/scarab 80x13
```
Average Scan Matching Score=1008.43
neff= 24.0941
Registering Scans:Done
^C[mapping-1] killing on exit
virtual GMapping::GridSlamProcessor::~GridSlamProcessor(): Start
virtual GMapping::GridSlamProcessor::~GridSlamProcessor(): Deleting tree
shutting down processing monitor...
... shutting down processing monitor complete
done
prorok@scarab44:~/ros/scarab_ws/src/scarab/scarab$
prorok@scarab44:~/ros/scarab_ws/src/scarab/scarab$
prorok@scarab44:~/ros/scarab_ws/src/scarab/scarab$
```

/home/prorok/ros/scarab_ws/src/scarab/scarab/scarab.launch http://scarab44.wifi:11311
```
grid_resolution: 0.020 sensor_sd: 0.020 subsample: 1
travel_distance: 0.200 travel_angle: 0.034
decay_duration: 200.000 decay_step: 40
[ INFO] [1432236878.116469999]: Connected to device with ID: H1200417
[ INFO] [1432236878.141194306]: Starting calibration. This will take up a few se
conds.
[ INFO] [1432236878.166148992]: Couldn't find transfrom from base_link to laser,
 using identity
[ INFO] [1432236879.166567959]: Calibration finished. Latency is: -0.0242
[ INFO] [1432236879.273783049]: Streaming data.
```

# Example: Navigate to Goal



```
[ INFO] [1432240776.833895837]: Setting goal: Frame:map_hokuyo, Position(-0.364,
 -1.061, 0.000), Orientation(0.000, 0.000, 0.000, 1.000) = Angle: 0.000

[ INFO] [1432240785.494624824]: Setting goal: Frame:map_hokuyo, Position(-0.810,
 -0.843, 0.000), Orientation(0.000, 0.000, 0.000, 1.000) = Angle: 0.000

[ INFO] [1432240800.648861342]: Setting goal: Frame:map_hokuyo, Position(-0.937,
 -0.318, 0.000), Orientation(0.000, 0.000, -0.669, 0.743) = Angle: -1.467
```

```
prorok@boga:~$
```

```
Publishing to /cmd_vel
Use w, a, s, d to drive, space to halt, and q to quit
Angular:  0.00
Linear:   0.00
```

```
started core service [/rosout]
```

# Actions

The `actionlib` package provides tools to create servers that execute long-running goals that can be **preempted**. It also provides a client interface in order to send requests to the server.

Action specification:

1. **Goal**: sent to an ActionServer by an ActionClient.
2. **Feedback**: provides server implementers a way to tell an ActionClient about the incremental progress of a goal
3. **Result**: sent from the ActionServer to the ActionClient upon completion of the goal (sent just once).

The action specification is defined using a `.action` file

Useful link:
http://wiki.ros.org/actionlib

# Goal to Action

1. Create a node that publishes messages to `goal` topic.
2. Create a node that subscribes to `goal` topic:
`…/scarab/hfn/scripts/goal_to_action.py`

This python program receives goal points from publisher node, and calls a callback function to transform it to an action:

```
rospy.Subscriber("goal", PoseStamped, callback)
```

The goal is processed in callback function, and sent to a MoveAction client:

```
# create client in main
client = actionlib.SimpleActionClient('move', MoveAction)
.
.
.
# send goal to action server
client.send_goal(goal)
```

## Using an Existing Map

We can create a map once and use it subsequently when operating in that environment (for localization navigation purposes, etc).
Use map_server package to save and load maps.

```
rosrun map_server map_saver -f mymapfilename map:=mymaptopic
```

Remap the topic name to match the topic name used by your subscriber.

Useful links:
http://wiki.ros.org/map_server#map_saver
http://wiki.ros.org/Remapping%20Arguments

## Simulation

For testing purposes, we can run code for the Scarab robot in a kinematic simulator. The code is identical to the one we would run on the actual hardware.

**Steps:**

1. run ROS with a local master URI
2. **`roslaunch scarab simulation.launch`**
3. **`rosrun rviz rviz`**
4. choose the appropriate displays to view (i.e., robot frame, map, laser)
5. **`roslaunch scarab hfn.launch`**
6. send waypoints to **`/goal`** topic

**DEMO**

# ROS-Swarmlet Interface V1

**Idea:**
Send a waypoint (or vector of waypoints) to the robot using the `hfn` package.

**Suggestion 1:**
Adapt the script …`/scarab/hfn/scripts/goal_to_action.py` so that goal points are received from a swarmlet (via accessors).

Instead of using a ROS publisher of `/goal` topic, grab the goal data from an accessor (from within this Python script).
Replace lines

```
rospy.Subscriber("goal", PoseStamped, callback)
rospy.spin()
```

with something like the following:

```
while AccessorDataAvailable():
  data = GetAccessorData()
  callback(data)
```

## ROS-Swarmlet Interface V2

**Idea:**
Send a waypoint (or vector of waypoints) to the robot using the `hfn` package.

**Suggestion 2:**
Use the script `…/scarab/hfn/scripts/goal_to_action.py` as it is.

Create your own ROS publisher node that sends messages to `/goal` topic using client libraries (c++ or python). This node represents the accessor-to-ROS interface.

# Appendix

# Dependencies

| | |
|---|---|
| accache | strace |
| openssl-server | ipython |
| ttf-dejavu | python-serial |
| libncurses-dev | git-core |
| libnl-dev | build-essential |
| libarmadillo-dev | python-yaml |
| libncurses5-dev | cmake |
| libcgal-dev | minicom |
| ros-indigo-desktop-full | iputils-arping |
| ros-indigo-openni2-launch | iputils-tracepath |
| ros-indigo-hokuyo-node | iputils-clockdiff |
| ros-indigo-joystick-drivers | cfengine2 |
| ros-indigo-navigation | console-common |
| ros-indigo-octomap-mapping | acpid |
| ros-indigo-gmapping | ifplugd |
| ros-indigo-octomap-rviz-plugins | batctl |
| python-rosinstall | batctl-dbg |
| python-rosdep | traceroute |
| iperf | olsrd |
| python-pip | olsrd-plugins |
| libgsl0-dev | nfs-common |
| libgsl0-dbg | rsync |
| libgsl0ldbl | subversion |

# Topics vs Services vs Actions

**Topics** should be used for *continuous data streams* (sensor data, robot state, ...).

**Services** should be used for *remote procedure calls that terminate quickly*, e.g. for querying the state of a node or doing a quick calculation. They should never be used for longer running processes, in particular processes that might be required to be preempted if exceptional situations occur. These processes should never change or depend on state to avoid unwanted side effects for other nodes.

**Actions** should be used for everything that moves the robot or that runs for a longer time such as perception routines that are triggered by some node and need a couple of seconds to terminate. The most important property of actions is that they can be *preempted,* and preemption should always be implemented cleanly by action servers. Another nice property of actions is that they can keep *state* for the lifetime of a goal, i.e. if executing two action goals in parallel on the same server, for each client a separate state instance can be kept since the goal is uniquely identified by its ID.

# Topics vs Services vs Actions

**Topics:** Continuous data flow. Data might be published and subscribed at any time independent of any senders/receivers. Many to many connection. Callbacks receive data once it is available. The publisher decides when data is sent.

**Service/Actionlib:** On-demand connection for one specific task. Service calls/ Actionlib tasks are processed when the client decides to request so. Tasks take time to complete.

**Service/Actionlib** are thus very similar and can actually be used interchangeably in their functionality. However, they serve different purposes:

**Service:** Simple blocking call. Mostly used for comparably fast tasks as requesting specific data. Semantically for processing requests.

**Actionlib:** More complex non-blocking background processing. Used for longer tasks like execution of robot actions. Semantically for real-world actions.