ورودی آسان به رآس

نویسنده:

جیسن ام اُ کین

مترجم:

زهرا بروجنی

تیرماه ۱۳۹۵

پیشگفتار مترجم:

فهرست مطالب

فـصل ۱ : مقدمه ۱

۱ – ۱ – چرا رآس؟ ۲

۱ – ۲ – چه انتظاری داریم. ۵

۱ – ۳ – قراردادها ۷

۱ – ۴ – برای اطلاعات بیشتر ۷

۱ – ۵ – در ادامه ۹

فـصل ۲ : برای شروع ۱۰

۲ – ۱ – نصب رآس ۱۱

۲ – ۲ – تنظیم اکانت تان ۱۳

۲ – ۳ – یک مثال کوچک با استفاده از turtlesim ۱۴

۲ – ۴ – پکیج ها ۱۶

۲ – ۵ – مَسترِ ۱۹

۲ – ۶ – نودها ۲۰

۲ – ۷ – تاپیک ها و پیام ها ۲۲

۲ – ۷ – ۱ – دیدن گراف ۲۳

۲ – ۷ – ۲ – پیام ها و نوع پیام ها ۲۵

۲ – ۸ – یک مثال بزرگ ۳۲

۲ – ۸ – ۱ – ارتباط به وسیله ی تاپیک خیلی به خیلی است ۳۳

۲ – ۸ – ۲ – نودها خیلی کم به هم وابسته اند. ۳۴

۲ – ۹ – چک کردن اشکالات ۳۵

۲ – ۱۰ – در ادامه خواهیم دید ۳۵

فـصل ۳ : نوشتن برنامه در رآس ۳۶

۳ – ۱ – ساختن یک فضای کاری و یک پکیج ۳۷

۳ – ۲ – سلام رآس! ۳۹

۳ – ۲ – ۱ – کمپایل کردن برنامه Hello ۴۱

۳ – ۲ – ۲ – اجرای برنامه ی hello ۴۴

۳ – ۳ – برنامه ی منتشر کننده ی پیام ۴۵

۳-۳-۱- منتشر کردن پیام ۴۶

۳ – ۳ – ۲ – انتشار در حلقه ۴۹

۳ – ۳ – ۳ – کمپایل کردن pubvel ۵۱

۳ – ۳ – ۴ – اجرای pubvel ۵۱

۳-۴- برنامه ی شنونده (ساب اسکرایبر) ۵۲
۳-۴-۱- کامپایل کردن و اجرای برنامه ی subpose ۵۷
۳-۵- در ادامه ۵۷

فصل ۴ : پیام های لاگ **۵۸**
۴-۱- سطح شدت ۵۹
۴-۲- یک برنامه ۶۰
۴-۳- ایجاد پیام های لاگ ۶۲
۴-۴- دیدن پیام های لاگ ۶۵
۴-۴-۱- کنسول ۶۶
۴-۴-۲- پیام ها در rosout ۶۷
۴-۴-۳- فایل های لاگ ۶۹
۴-۵- فعال کردن و غیرفعال کردن پیام های لاگ ۷۰
۴-۶- در ادامه ۷۳

فصل ۵ : گرافِ اسم های منابع **۷۴**
۵-۱- نام های جهانی (Global names) ۷۵
۵-۲- اسم های نسبی ۷۶
۵-۳- اسم های خصوصی ۷۸
۵-۴- اسم های مستعار ۷۹
۵-۵- در ادامه ۸۰

فصل ۶ : فایل های لانچ **۸۱**
۶-۱- استفاده از فایل های لانچ ۸۲
۶-۲- ساختن فایل لانچ ۸۵
۶-۲-۱- کجا باید فایل ها ی لانچ را قرار دهیم ۸۵
۶-۲-۲- مواد اولیه ۸۶
۶-۳- نگاشت اسم ها ۹۲
۶-۳-۱- ایجاد یک نگاشت ۹۲
۶-۳-۲- معکوس کردن یک لاک پشت ۹۳
۶-۴- المان های دیگری از فایل لانچ ۹۸
۶-۴-۱- اضافه کردن فایل های دیگر ۹۹
۶-۴-۲- آرگومان های لانچ ۱۰۰
۶-۴-۳- ساختن گروه ها ۱۰۲
۶-۵- در ادامه ۱۰۳

فصل ۷: پارامترها **۱۰۴**
۷-۱- دسترسی به پارامترها از کامند لاین ۱۰۵
۷-۲- مثال: پارامترها در turtlesim ۱۰۷
۷-۳- دسترسی به پارامتر از C++: ۱۰۹
۷-۴- تنظیم پارامترها در فایل های لانچ ۱۱۳
۷-۵- در ادامه ۱۱۵

فصل ۸: سرویس ها **۱۱۶**
۸-۱- واژه شناسی سرویس ها ۱۱۷
۸-۲- پیدا کردن و فراخوانی سرویس با command line ۱۱۸
۸-۳- یک برنامه درخواست کننده ۱۲۲
۸-۴- برنامه سرور ۱۲۷
۸-۴-۱- اجرا کردن و بهبود بخشیدن برنامه سرور ۱۳۰
۸-۵- در ادامه ۱۳۱

فصل ۹: ضبط و بازپخش پیام ها **۱۳۲**
۹-۱- ذخیره کردن و دوباره پخش کردن فایل های bag ۱۳۳
۹-۲- مثال: یک bag از مربع ها ۱۳۵
۹-۳- Bags درون فایل های لانچ ۱۳۹
۹-۴- در ادامه ۱۴۰

فصل ۱۰: جمع‌بندی **۱۴۱**
۱۰-۱- قدم بعدی ۱۴۲
۱۰-۲- نگاهی به آینده ۱۴۴

فهرست اشکال

فهرست اشکال . . . . . و
شکل (۱-۱) ارتباط بین فصول ٦
شکل (۲-۱) پنجره turtlesim، قبل و بعد از ترسیم چند حرکت ۱۵
شکل (۲-۲)rqt_graph گراف مثال turtlesim را نشان می دهد. نودهای دیباگ مانند rosout به طور پیش فرض حذف شده اند. ۲۳
شکل (۲-۳) نمودار کامل turtlrsim شامل نودهایی که rqt_graph به عنوان نودهای اشکال زدایی طبقه بندی کرده است. ۲۳
شکل (۲-۴) گراف turtlesim، که همه ی تاپیک ها، شامل تاپیک هایی بدون منتشر کننده یا شنونده، را به عنوان یک شیء جدا نشان می دهد. ۲۵
شکل (۲-۵) گرافی پیچیده از رآس، با دو نود turtlesim به نام های A و B و نودهای کنترل از راه دور به نام های C و D. ۳۳
شکل (۳-۱) عکس العمل لاک پشت turtlesim به دستور سرعت تصادفی از pubvel ۵۲
شکل (۴-۱)GUI (رابط گرافیکی) برای rqt_console ٦۸
شکل (۴-۲)GUI برای rqt_logger_level. ۷۲
شکل (۶-۱) نودها و تاپیک ها (به ترتیب در بیضی و مستطیل) که به وسیله ی doublesim.launch ایجاد شده اند. ۹۰
شکل (۶-۲) نتیجه ی گراف رآس از تلاشی نادرست برای استفاده از reverse_cmd_vel برای معکوس کردن لاک پشت turtlesim. ۹٦
شکل (۶-۳) نتیجه گراف درست از reversed.launch. المان remap این امکان را فراهم می کند که نود به درستی ارتباط برقرار کند. ۹۷
شکل (۷-۱) قبل (چپ) و بعد (راست) تغییر رنگ پشت زمینه ی نود turtlesim. ۱۰۹
شکل (۸-۱) نتیجه ی اجرای pubvel_toggle با مقداری چرخش، با فراخواندن دستی /toggle_forward. ۱۳۰
شکل (۹-۱) گراف نودها و تاپیک ها در حین اجرای rosbag record ۱۳٦
شکل (۹-۲) گرافی از نودها و تاپیک ها حین اجرای rosbag play ۱۳۷
شکل (۹-۳) (چپ) یک لاک پشت در پاسخ به دستور حرکت از draw_square. حرکت ها به وسیله ی rosbag ذخیره شده است. (راست) با بازپخش کردن فایل bag, ما دنباله ای از پیام ها را برای لاک پشت می فرستیم. ۱۳۷

فـصل ۱ : مقدمه

در این بخش مزایای رآس و سرفصل بخش های بعدی را بیان می کنیم.

۱-۱- چرا رآس؟

محققین رباتیک در سال های اخیر پیشرفت شگرفی داشته است. سخت افزار های قابل اعتماد و ارزان رباتیک ، از ربات های متحرک بر زمین تا کوادروتور و ربات های انسان نما، به صورت گسترده در دسترس هستند. و البته محققین رباتیک الگوریتم های زیادی برای خودمختار نمودن ربات ها توسعه داده اند.

با این وجود این پیشرفت سریع، در زمینه ی نرم افزار چالش های زیادی در رباتیک وجود دارد. این کتاب یک پلت فرم نرم افزار به نام سیستم عامل رباتیک (رآس) را معرفی می کند. هدف از رآس در سایت رسمی اش به صورت زیر عنوان شده است:

رآس یک سیستم عامل رایگان با قابلیت کار بر روی چند سیستم عامل برای ربات است و سرویس هایی که شما از یک سیستم عامل انتظار دارید را فراهم می کند. رآس شامل شبیه ساز سخت افزار- کنترل سطح پایین- انتقال پیام های بین پردازش ها و توابع- مدیریت پکیج ها می باشد. بعلاوه رآس شامل ابزار ها و کتابخانه هایی برای کمپایل کردن و نوشتن و اجرا کردن کد در چند کامپیوتر است. [۱]

این توصیف دقیق است و کاملاْ بیان می کند که رآس جایگزینی برای بقیه سیستم عامل ها نیست بلکه در کنار آنها کار می کند. پس فایده ی واقعی رآس چیست؟ یادگرفتن رآس به زمان و انرژی زیادی نیاز دارد پس باید مطمئن باشیم که ارزش سرمایه گذاری این را دارد. در ادامه فواید توسعه نرم افزارهای رباتیک که رآس می تواند در این زمینه به ما کمک کند را بیان می کنیم.

توزیع محا سبه : امروزه بسیاری از ربات ها برا ساس نرم افزارهایی هستند که به و سیله ی چند کامپیوتر اجرا می شود. برای مثال :

بعضی از ربات ها چند کامپیوتر را حمل می کنند که هر کدام یکی از محرک ها یا سنسورهای ربات را کنترل می کند.

این ایده ی بسیار خوبی است که حتی با یک کامپیوتر، نرم افزار به بخش های کوچک و مستقلی،

[۱] http://wiki.ros.org/ROS/Introduction

که با همکاری یکدیگر هدف نهایی را تحقق ببخشند، تقسیم شود. این روش "پیچیدگی در ترکیب"[1] خوانده می شود.

وقتی چند ربات برای انجام کاری مشترک با هم همکاری می کنند اغلب لازم است که برای راهبری کارهایشان با هم ارتباط برقرار کنند.

انسان ها معمولاْ دستوری را به وسیله کامپیوتر، لپتاپ یا موبایل به ربات ها می فرستند در این مورد می توان ارتباط انسان را نیز به عنوان یک نرم افزار ربات در نظر گرفت.

مساله ی مشترک در همه ی موارد بالا ایجاد ارتباط بین نرم افزار های مختلف است که ممکن ا ست در یک کامپیوتر یا چند کامپیوتر اجرا شوند. رآس دو مکانیزم ساده برای این نوع ارتباط ها فراهم می کند که در بخش های سه و هشت به آن می پردازیم.

استفاده دوباره از کدها: توسعه سریع رباتیک به دلیل جمع آوری الگوریتم های خوب برای وظایف معمول ربات ها مانند جهت یابی، مسیریابی و نقشه کشی و... است. و در واقع وجود چنین الگوریتم های زمانی مفید خواهد بود که راهی برای پیاده سازی آنها با مفهوم جدید- بدون نیاز به پیاده سازی هر الگوریتم برای هر سیستم جدید - وجود داشته باشد. رآس برای جلوگیری از چنین مشقتی می تواند حداقل از دو راه زیر کمک نماید.

- پکیج های استاندارد رآس الگوریتم های رباتیک مهم زیادی که پایدار و قابل عیب یابی و رفع عیب هستند را فراهم می کند.
- پیام های رآس در واقع استانداری برای ارتباط بین نرم افزارها هستند- یعنی ارتباط رآس با هر دو طرف سخت افزار و نرم افزارهایی که الگوریتم های پیشرفته را پیاده سازی کرده اند، کاملا فراهم شده است. برای مثال در سایت رآس لیست صدها پکیج وجود دارد.[2] این نوع ارتباط های استاندارد شده نیاز به کدهای کمکی برای مرتبط کردن قسمت ای مختلف نرم افزار را بسیار کاهش می دهد.

به این ترتیب بعد از یادگرفتن رآس می توان بی شتر روی ایده های جدید تمرکز کرد تا اینکه چرخ را دوباره بسازیم.

ت ست سریع: یکی از دلایلی که گسترش نرم افزار رباتیک از گسترش بقیه قسمت ها چالش برانگیزتر است، وقت گیر بودن تست و وپیدا کردن خطاهاست. ربات واقعی ممکن است همیشه در

---

[1] complexity via composition

[2] http://www.ros.org/browse

دسترس نباشد، و وقتی در دسترس است پردازش ها بسیار کند و خسته کننده هستند. رآس دو ابزار مؤثر برای این مشکل فراهم می کند.

- سیستم های خوب طراحی شده رآس کنترل مستقیم سطح پایین سخت افزار را از پردازش سطح بالا و بخش تصمیم گیری جدا می کند. به دلیل این جدایی ما می توانیم به صورت موقت کنترل های سطح پایین و سخت افزارهای مرتبط با آن را به وسیله شبیه سازجایگزین کنیم تا الگریتم های سطح بالا را تست کنیم.
- رآس همچنین امکان ذخیره کردن و بازپخش کردن داده های سنسورها و پیام ها را فراهم میکند. این امکانات یعنی ما می توانیم از زمان بهره ی بیشتری ببریم چون با ثبت کردن داده های سنسور می توانیم آنها را بارها دوباره اجرا کنیم و با روش های مختلف آنها را پردازش کنیم. به این ابزار در رآس بگ (bag) می گوییم و در واقع رآس بگ (rosbag) ابزاری است برای رکورد کردن و دوباره اجرا کردن که در بخش ۹ به آن می پردازیم.

مساله مهم اینجا ست که تفاوت اطلاعات ذخیره شده و سنسور واقعی ناچیز است. چون ربات واقعی، شبیه ساز، و دوباره اجرا کردن بگ فایل، هر سه یک نوع دیتای ارتباطی ایجاد می کند. و کدتان لازم نیست در دو مد مختلف کار کند و حتی لازم نیست مشخص کنیم با ربات واقعی در ارتباط است یا با دیتای ذخیره شده (بگ).

البته رآس تنها پلت فرمی نیست که این قابلیت ها را فراهم می کند. اما آنچه را رآس را متمایز می کند جامعه گسترده رباتیکی است که از آن استفاده می کند و آن را پشتیبانی می کند. این پشتیبانی وسیع منطقا رآس را ادامه دار، گسترش پذیر و قابل بهبود در آینده نشان می دهد.

رآس چه چیزهایی نیست ... در ادامه مواردی که در مورد رآس درست نیست را بیان می کنیم.

- رآس یک زبان برنامه نویسی نیست. بلکه در این کتاب توضیح می دهیم چگونه به زبان C++[۱] در رآس برنامه نویسی کنیم. گرچه می توان به زبان های پایتان[۲]، جاوا[۳] و لیسپ[۴] و دیگر زبان ها[۵] هم می توان در این محیط برنامه نویسی کرد.

[۱] http://wiki.ros.org/roscpp

[۲] http://wiki.ros.org/rospy

[۳] http://wiki.ros.org/rosjava

[۴] http://wiki.ros.org/roslisp

[۵] http://wiki.ros.org/ClientLibraries

- رآس فقط یک کتابخانه نیست. بلکه سرویس های مرکزی و ابزار دستوری (command-line) و گرافیکی و ساختن (build) را هم فراهم میکند.
- رآس یک محیط توسعه مجتمع(Integrated Development Environment) نیست. اگرچه رآس هیچ محیطی را تجویز نمی کند اما می تواند با بیشتر IDE[1] معمول استفاده شود. البته به نظر شخصی نویسنده استقاده از دستورات command line بدون IDE کاملا منطقی است.

۱-۲- چه انتظاری از رآس داریم

هدف از این کتاب مروری بر مفاهیم و تکنیک هایی است که برای نوشتن یک کد رآس نیاز دارید. این هدف محدودیت هایی را برمحتوای این کتاب اعمال می کند.

- این کتاب معرفی زبان برنامه نویسی نیست. ما در مورد پایه های برنامه نویسی بحث نمی کنیم. در این کتاب فرض شده است شما به اندازه کافی توانایی خواندن و نوشتن و فهمیدن کد C++ را دارید
- این کتاب یک رفرنس نیست. اطلاعات دقیقی در مورد رآس شامل راهنمای استفاده از رآس به صورت آن لاین موجود است.[2] این کتاب جایگزینی برای منابع دیگر نیست.[3] در عوض ما قسمت هایی از رآس را انتخاب کرده ایم که به نظر نویسنده شروع خوبی برای استفاده از رآس است.
- این کتاب، کتاب معرفی الگوریتم های رباتیک نیست. یاد گرفتن رباتیک و به خصوص الگوریتم های کنترل ربات های خودمختار می تواند کاملا جذاب باشد. الگوریتم بسیاری برای حل بخش های مختلف این مسئله (کنترل ربات های خودمختار) بوجود آمده است. این کتاب هیچ کدام از این الگوریتم ها را درس نمی دهد (البته شما باید آنها را به هر حال یاد بگیرید!!). تمرکز ما بر آموزش ابزارهای رآس برای سهولت در پیاده سازی و تست این الگوریتم هاست.

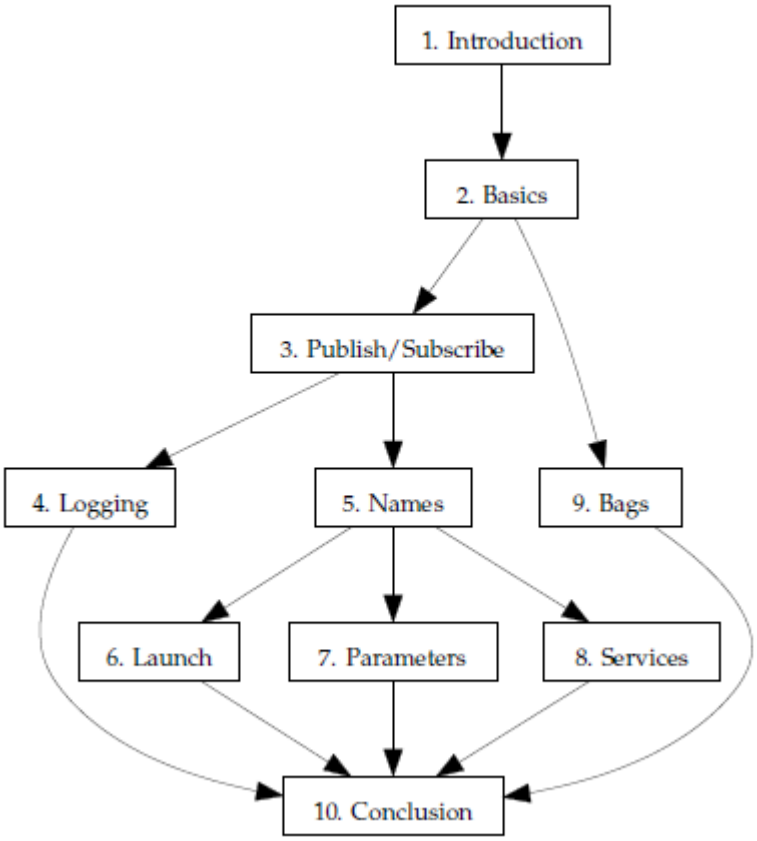
[1] http://wiki.ros.org/IDEs

[2] http://wiki.ros.org/ROS/Tutorials

[3] http://wiki.ros.org/APIs

فصل ها و ارتباط آنها: شکل ۱-۱ ساختار کتاب را نشان می دهد. فصل ها با مستطیل و ارتباط بین ف صول با بردار ن شان داده شده اند. مطالعه این کتاب با هر ترتیبی که با محدودیت های ذیل هماهنگی داشته باشد منطقی است.

خواندن کتاب برای دانشجویان و محققان و علاقه مندان که می خواهند سریع با رآس آ شنا بشوند مفید با شد. ما فرض کرده ایم که شما با محیط لینوکس آ شنا هستید و با C++ برای نو شتن نرم افزار و کنترل ربات آ شنا هستید و به طور کلی فرض بر این است که شما از اوبونتو ۱۴,۰۴ و bash فایل ها استفاده می کنید.



شکل (۱-۱) ارتباط بین فصول

۱-۳- قراردادها

در طول این کتاب ما تلاش می کنیم مشـــکلات معمول را پیش بینی کنیم. این نوع اخطارها، که واقعا ارزش توجه دارند، بخصـــوص وقتی که کار آن طور که می خواهیم انجام نمی شـــود، با پس زمینه ی صورتی و حاشیه مضاعف نشان داده می شود.

این کادر نشان دهنده ی یک منبع مشکل متداول است.

به علاوه، بعضـی بخش ها شـامل توضـیحاتی اسـت که برای برخی خواننده ها ممکن اسـت جالب با شد، اما برای سایرین درک مفاهیمش الزامی نی ست. این موارد با پ شت زمینه ی آبی و حا شیه خط چین مشخص شده اند.

کادر آبی و خط چین شامل توضیحاتی است که می توان به سرعت از آنها گذر کرد، به ویژه در دور اول مطالعه.

۱-۴- برای اطلاعات بیشتر

همان طور که در بالا ا شاره شد این کتاب سعی ندارد یک مرجع جامع برای رآس با شد. شما برای انجام کارهای دلخواهتان حتما به اطلاعات جزیی تری نیاز دارید. خوش بختانه اطلاعات آن لاین زیادی برای رآس وجود دارد.

- مهمترین آنها، گسترش دهندگان رآس مستندات وسیعی را، که شامل خودآموزهای رآس نیز می باشـد، نگهداری می کنند.[۱] این کتاب شـامل لینک های زیادی به این نوشـته ها اسـت که در پاورقی آورده شـده اند. اگر شـما نسـخه الکترونیکی این کتاب را می خوانید، شـما باید بتوانید روی لینک ها به صـورت مسـتقیم کلیک کنید تا مسـتقیم روی مرورگر شما باز شود.
- وقتی موردی پیش بینی نشده پیش آمد، می توانید به فروم (forum) سوال و جواب سایت رآس مراجعه کنید. [۲]

[۱] http://wiki.ros.org

[۲] http://answers.ros.org

- همچنین شما می توانید عضو لیست ایمیل های رآس بشوید، که بتوانید اطلاعیه های آن را دنبال کنید.[۱]

در اینجا دو مورد مهم از جزییاتی که می توانید در منابع کمک آموزشی آن لاین ببینید که می توانند کمک کننده باشد، اما به صورت کامل توضیح داده نشده اند، آورده شده است.

توزیع ها : ویرایش های رآس، توزیع های رآس نامیده می شوند، و با صفاتی که به ترتیب الفبای انگلیسی هستند نام گذاری می شوند.[۲] (این اسم گذاری مانند بقیه پروژه های بزرگ نرم افزاری مانند ابونتو و اندروید است.) در حال حاضر ویرایش طولانی مدت ایندیگو (indigo) است. نسخه بعدی jade است که در ماه می ۲۰۱۵ منتشر شده است[۳]. نسخه های قبلی شامل
Indigo, hydro, groovy, fuerte, electric, diamondback, C turtle, box Turtle
می باشد. نام این توزیع ها در این کتاب بارها در جاهای مختلف استفاده خواهد شد.

در این کتاب فرض شده است شما از نسخه indigo استفاده می کنید

اگر به هر دلیلی شما از نسخه ی قدیمی تر hydro استفاده می کنید، تقریبا همه ی محتوی این کتاب بدون تغییر قابل تعمیم است.

این مورد در مورد ویرایش groovy هم صادق است. به جز یک مورد مهم: در توزیع های جدیدتر از groovy و (در نتیجه در این کتاب) دستورهای سرعت برای turtle-sim به صورت یک پیام استاندارد و یک نام تایپک استاندارد برای استفاده دیگر ربات ها تعریف شده است.

در واقع اگر از نسخه ی groovy استفاده می کنید باید تغییرات کوچکی را اعمال کنید:

- وقتی پکیج های پیش نیاز را اضافه می کنید (صفحه ی ۴۴)، شما به پکیج turtlesim به جای geometry_msgs نیاز دارید.
- هیدر فایل مرتبط (صفحه ۴۶) turtlesim/Velocity.h است به جای geometry_msgs/Twist.h.

نوع پیام turtlesim/Velocity دارای دو فیلد است که linear (خطی) و angular (چرخشی) است. این فیلدها مانند فیلدهای linear.x و angular.z در geometry_msgs/Twist است. این تغییرات باید در command line (صفحه ۲۹) و کد C++ (صفحه ۴۵) اعمال شود.

[۱] http://lists.ros.org/mailman/listinfo/ros-users

[۲] http://wiki.ros.org/Distributions

[۳] http://wiki.ros.org/jade

ساختن سیستم ها: بعد از توزیع groovy، رآس تغییرات گسترده ای برای کمپایل کردن نرم افزار انجام داد. قبل از توزیع groovy سیستم rosbuild استفاده می شد. اما الان catkin جایگزین rosbuild شده است. این بسیار مهم است که شما این تغییر را بدانید چون تعدادی از این منابع آموزشی براساس نوع توزیع دسته بندی شده اند، براساس اینکه rosbuild یا catkin استفاده می کنند. این کتاب catkin را توضیح می دهد، اما ممکن است مواردی باشد که در آنها rosbuild انتخاب بهتری باشد.[1]

۱-۵- در ادامه

در فصل بعد، ما کار کردن با رآس را شروع می کنیم و بعضی از مفاهیم و ابزارها را یاد می گیریم.

[1] http://wiki.ros.org/catkin_or_rosbuild

فـصل ۲ : برای شروع

در حین نصب کردن رآس، بعضی از مفاهیم ابتدایی رآس را معرفی می کنیم و با محیط کار رآس بیشتر آشنا می شویم.
قبل از وارد شدن به جزئیات نوشتن یک نرم افراز با استفاده از رآس، بهتر است که رآس را نصب و راه اندازی کنیم و کمی از ایده های اولیه ای که در رآس استفاده شده است را بفهمیم. این فصل پایه ی کار است. بعد از بررسی سریع نصب رآس و تنظیم اکانت کاربری خود برای استفاده از آن، ما سیستم کاری رآس را بررسی می کنیم (به خصوص با مثال شبیه ساز لاک پشت) و چگونه ارتباط با سیستم را با استفاده از ابزارهای کامند لاین یاد می گیریم.

۲-۱- نصب رآس

قبل از هرکاری با رآس، طبیعتا ما باید مطمئن شویم که رآس روی کامپیوترتان نصب شده است. اگر شما روی کامپیوتری که رآس ـ شامل پکیج ros-indigo-turtlesim ـ روی آن نصب شده است کار می کنید، می توانید مستقیماً از بخش ۲-۲ شروع کنید. مراحل نصب در سایت رآس به صورت کاملاً سرراست نوشته شده است.[۲] [۱] در اینجا ما خلاصه ای از مراحل ضروری را ذکر می کنیم.

اضافه کردن مکان مخزن رآس (ROS repository): به عنوان روت (به عنوان کاربر اصلی)، فایل زیر را ایجاد کنید

/etc/apt/sources.list.d/ros-latest.list

این خط را در فایل بنویسید

deb http://packages.ros.org/ros/ubuntu trusty main

این خط مخصوص اوبونتو ۱۴,۰۴ Ubuntu 14.04 است، که نام آن trusty است. اگر شما اوبونتو ۱۳و۱۰ استفاده می کنید، شما می توانید saucy را به جای trusty جایگزین کنید.

برای ویرایش های دیگر اوبونتو ، چه قبل تر چه بعدتر، پکیج های کمپایل شده برای ویرایش ایندیگو رآس پشتبانی نمی شوند. به هر حال برای ویرایش های جدیدتر از ۱۴,۰۴ نصب رآس از منبع[۳] یک گزینه ی منطقی است

[۱] http://wiki.ros.org/ROS/Installation

[۲] http://wiki.ros.org/indigo/Installation/Ubuntu

[۳] http://wiki.ros.org/indigo/Installation/Source

اگر شما از ویرایش اوبونتویی که استفاده می کنید مطمئن نیستید، می توانید با استفاده از کامند زیر آن را پیدا کنید.

lsb_release -a

خروجی نامِ کد و عدد اوبونتو را باید نشان دهد.

نصب کلید احراز هویت بسته: قبل از نصب پکیج رآس، شما باید کلید احرازهویت پکیج هایش را به دست آورید. پس اول کلید را دانلود کنید:

wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key

اگر این دستور به درستی کار کرد، شما یک فایل باینری کوچک به نام **ros.key** خواهید داشت. بعد شما باید مدیریت پکیج های سیستم را برای استفاده از کلید تنظیم کنید.

sudo apt-key add ros.key

بعد از تکمیل این بخش (apt-key باید " OK " را نشان دهد)، شما می توانید ros.key را پاک کنید.

دانلود لیست پکیج ها: بعد از اینکه مخازن تنظیم شدند، شما می توانید آخرین لیست پکیج های در دسترس را به روش معمول به دست آورید:

sudo apt-get update

توجه داشته باشید که این دستور نه تنها مخازن جدید رآس را، بلکه همه ی مخازن سیستم تان را، به روز رسانی می کند.

نصب پکیج های رآس: اکنون ما می توانیم نرم افزار رآس را نصب کنیم. ساده ترین راه، نصب کامل هسته سیستم رآس است:

sudo apt-get install ros-indigo-desktop-full

اگر شما فضای دیسک آزاد زیادی دارید - چند گیگابایت باید کافی باشد- این پکیج بی شک بهترین انتخاب است. اگر شما به فضای دیسکتان نیاز دارید، پکیج های جایگزین فشرده تری موجود است، شامل ros-indigo-desktop و ros-indigo-ros-base، که بعضی از پکیج ها و ابزارها را برای کاهش فضای دیسک مورد نیاز حذف کرده است.

نصب شبیه ساز لاک پشت turtlesim: در این کتاب ما بکرات به این شبیه ساز ساده مراجعه می کنیم تا چگونه کارکرد دستورات را نشان دهیم. اگر شما می خواهید مثال ها را دنبال کنید ـ که ما پیشنهاد می کنیم ـ شما باید turtlesim را نصب کنید. با استفاده از دستور زیر:

sudo apt-get install ros-indigo-turtlesim

تنظیم سیستم فراگیر rosdep: بعد از نصب پکیج های رآس شما باید دستور زیر را اجرا کنید:

sudo rosdep init

این مقداردهی اولیه را یک بار انجام می دهیم؛ وقتی رآس درست کار کند، بسیاری از کاربران دیگر نیاز به rosdep ندارند.

همانطور که از اسمش معلوم است، هدف از این دستور تنظیم rosdep است، که یک ابزار برای بررسی و نصب وابستگی های پکیج مستقل از سیستم عامل است. [۱] به طور مثال، در اوبونتو rosdep به صورت یک واسط (front end) بین کاربر و apt-get عمل می کند. ما rosdep را به صورت مستقیم استفاده نمی کنیم، بلکه ما از تعدادی ابزار که آن را از پشت صحنه فرا می خواند استفاده می کنیم. این ابزارها در صورتیکه rosdep درست تنظیم نشده باشد، خیلی ناکارآمد خواهند بود.

در دستورکار آنلاین ابزاری به نام rosinstall ذکر شده است، که وظیفه اش نصب رآس از منبع است. [۲ ۳] نرم افزارهایی که در این کتاب نیاز داریم در پکیج های deb اوبونتو موجود است، در نتیجه به rosinstall نیازی نیست.

۲-۲- تنظیم اکانت تان

چه رآس روی کامپیوترتان نصب بوده یا شما آن را روی کامپیوتر نصب کرده اید، هر کاربر باید دو قدم مهم تنظیم برای استفاده از رآس انجام دهد.

تنظیم rosdep **برای اکانت کاربر:** شما اول باید rosdep را روی اکانت تان با استفاده از دستور زیر مقداردهی اولیه کنید:

```
rosdep update
```

این دستور تعدادی فایل را در شاخه home شما در زیر شاخه به نام ros. ذخیره می کند. به صورت معمول این دستور فقط یکبار باید اجرا شود.

توجه داشته باشید برخلاف rosdep init در بالا، دستور rosdep update را باید به عنوان یک کاربر معمولی اجرا کنید و نیازی به sudo نیست.

[۱] http://wiki.ros.org/rosdep

[۲] http://wiki.ros.org/rosinstall

[۳] http://www.ros.org/doc/independent/api/rosinstall/html/

تنظیم متغیرهای محیطی: رآس به تعدادی متغییرهای محیطی برای تعیین فایل ها وابسته است. برای تنظیم این متغییرهای محیطی، شما باید متن setup.bash که رآس فراهم کرده است را اجرا کنید. با استفاده از دستور source زیر[1]

```
source /opt/ros/indigo/setup.bash
```

بعد با استفاده از دستور زیر می توانید درست تنظیم شدن متغییرهای محیطی را تأیید کنید

```
export | grep ROS
```

اگر همه چیز به درستی کار کرد، شما باید تعداد انگشت شماری از مقادیر (مقادیری برای متغییرهای محیطی مانند ROS_DISTRO و ROS_PACKAGE_PATH) در خروجی دستور ببینید. اگر setup.bash هنوز اجرا نشده باشد خروجی این دستور خالی خواهد بود.

اگر خطای " command not found " را بعد از دستورات رآس که بعداً در این بخش معرفی می شود دریافت کردید، بیشتراوقات دلیل این است که شما setup.bash را در ترمینال (shell) جاری اجرا نکرده اید.

توجه کنید گرچه که قدم های بالا تنها روی ترمینال جاری اجرا می شود، به سادگی می توانید در هر ترمینالی که می خواهید دستورات رآس را اجرا کنید، درابتدا دستور source را اجرا کنید. با این وجود، این موضوع می تواند ناراحت کننده و براحتی قابل فراموشی باشد، بخصوص وقتی شما می خواهید سیستم های زیادی با دستورات مختلف و به صورت همزمان در ترمینال های جداگانه اجرا کنید.

بنابراین، شما می توانید اکانت خود را طوری تنظیم کنید که متن setup.bash را هر دفعه برای ترمینال جدید به صورت خودکار اجرا کند. برای انجام این مورد، فایل bashrc. در شاخه **home** را تغییر دهید و دستور source را به آخر فایل اصافه کنید.

که علاوه بر تنظیم متغییرهای محیطی، متن setup.bash توابع bash برای پیاده سازی تعدادی از دستورات را تعریف می کند، شامل roscd و rosls، که در زیر معرفی می شوند. این توابع در پکیج rosbash تعریف می شوند.[2]

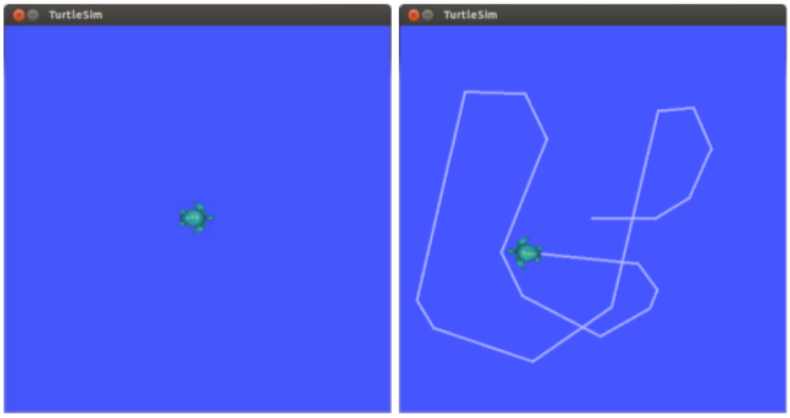
۲-۳- یک مثال کوچک با استفاده از turtlesim

قبل از آنکه جزیئات کارکرد رآس را مورد بررسی قرار دهیم، باید با یک مثال شروع کنیم. این مثال سریع چند هدف مختلف را دنبال می کند: به شما کمک می کند که مطمئن شوید رآس به

[1] http://wiki.ros.org/rosbash

[2] http://wiki.ros.org/rosbash

درستی نصب شده است، شبیه ساز لاک پشت turtlesim را معرفی می کند[1] ، که در بسیاری از دستورکارهای آن لاین و این کتاب از آن استفاده شده است، و همچنین یک سیستم کاربردی (البته بسیار ساده) فراهم می کند که ما در بخش های آینده به آن مراجعه خواهیم کرد.



شکل (۲-۱) پنجره turtlesim، قبل و بعد از ترسیم چند حرکت.

شروع turtlesim: در سه ترمینال جداگانه، سه دستور زیر را اجرا کنید

```
roscore
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

سه ترمینال جداگانه باعث می شود که بتوانیم سه دستور را به صورت همزمان اجرا کنیم. اگر همه چیز درست کار کند شما باید یک پنجره گرافیکی مانند قسمت چپ شکل ۲,۱ ببینید. این پنجره یک ربات شبیه سازی شده به صورت لاک پشت که در یک محیط مربع شکل فعال است را نشان می دهد. (شکل لاک پشت ممکن است متفاوت باشد و شبیه ساز آن را از بین کلکسیون لاک پشت ها "mascot" برای هر ویرایش رآس انتخاب می کند.) اگر روی سومین ترمینال (آنکه دستور turtle_teleop_key اجرا کردید) تمرکز کنید و با فشار دادن کلید های بالا، پایین، چپ و راست، لاک پشت متناسب با دستور شما حرکت خواهد کرد، و ردی از حرکت خود برجای می گذارد.

> 🖳 اگر لاک پشت در پاسخ به فشار دادن کلید شما حرکت نکرد، مطمئن شوید که ترمینال turtle_teleop_key حساس به ورودی است، برای مثال درون آن کلیک کنید. شما ممکن است نیاز داشته باشید که پنجره ها را طوری بچنید که روی این ترمینال تمرکز کنید در حالیکه پنجره شبیه ساز را نیز مشاهده می کنید.

[1] http://wiki.ros.org/turtlesim

ایجاد لاک پشت های مجازی که خط می کشند به خودی خود هیجان انگیز نیست.[1] گرچه این مثال که اکنون در پشت صحنه اجرا شد به اندازه کافی می تواند بسیاری از ایده های اصلی سیستم رآس را روشن کند.
شما باید این سه ترمینال را باز نگه دارید، چون در مثال های بخش های بعدی راه های دیگری برای ارتباط با این سیستم نشان خواهیم داد.

۲-۴- پکیج ها

همه نرم افزارهای رآس به صورت پکیج تنظیم شده اند. یک پکیج رآس یک مجموعه منسجم از فایل رآس است، به طور کلی شامل هر دوی فایل های اجرایی و پیشتیبانی، که در خدمت یک هدف خاص هستند. در این مثال، ما دو فایل اجرایی turtlesim_node و turtle_teleop_key را استفاده کردیم، که هر دو از اعضای پکیج turtlesim هستند.

مواظب تفاوت بین پکیج های رآس و پکیج های مورد استفاده ی مدیریت پیکج های سیستم عاملتان باشید، مانند پکیج های deb که در اوبونتو استفاده می شوند. مفاهیم مشابه اند و نصب پکیجی از deb ممکن ا ست یک یا چند پکیج از رآس را به نصب شما ا ضافه کند، اما این دو با هم یکی نیستند

اغراق نیست که گفته شود تمام نرم افزارهای ROS بخشهایی از پکیج های مختلف هستند. نکته مهم این است که این موضوع شامل برنامه های جدید که شما ایجاد می کنید هم هست. در بخش ۳,۱ ساختن پکیج های جدید را فراخواهیم گرفت. در این میان، رآس چند دستور برای تعامل با پکیج های نصب شده فراهم می کند.

لیست و مکان پکیج ها: شما می توانید یک لیست از پکیج های نصب شده با استفاده از دستور زیر به دست آورید: [2] [3]

rospack list

در سیستم نویسنده، این دستور لیستی از ۱۸۸ پکیج ایجاد کرد.
هر پکیج با یک بیانیه (فایل توصیفات) به نام package.xml معرفی می شود. این فایل جزیئاتی مانند اسم، ویرایش، نگهدارنده، وابستگی ها را در مورد پکیج مشخص می کند. شاخه شامل package.xml شاخه پکیج نامیده می شود. (در واقع این تعریف پکیج رآس است: هر شاخه شامل

[1] برای مثال نویسنده درست کردن لاک پشتی که روی صفحه کامپیوتر ردی برجای بگذارد را اولین بار حدود سال ۱۹۸۷ انجام داده است.

[2] http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem

[3] http://wiki.ros.org/rospack

فایلی به نام package.xml که رآس پیدا بکند شاخه پکیجی است.). بیشتر فایل های پکیج در این شاخه ذخیره می شوند.

یک استثناء مهم این است که ـــ بیشتر پکیج هاـــ مخصوصا آنهایی که برای استفاده از سیستم کمپایل catkin به روز رسانی شده اندـــ فایل اجرایی کمپایل شده در شاخه پکیج ذخیره نمی شوند، بلکه با یک سلسه مراتبی استاندارد شده ی دیگری در پوشه ی جداگانه ای ذخیره می شوند. برای پکیج هایی که به وسیله ی apt-get نصب شده اند، این سلسه مراتب در مسیر /opt/ros/indigo است. فایل اجرایی در زیر شاخه ی lib در این مسیر ذخیره شده است. وقتی به آنها نیاز است، رآس این فایل ها را با جستجو در شاخه های لیست شده در متغییر محیطی CMAKE_PREFIX_PATH، که به صورت خودکار در setup.bash مشخص شده اند، پیدا می کند. این نوع دسته بندی بیرون از منبع یکی از تغییرات بنیادی به وسیله ی catkin در ویرایش groovy به بعد در مقایسه با ویرایش های fuerte و قبلتر است. به صورت کلی، با همه ی این اتفاقات در پشت صحنه، ما می توانیم به رآس برای پیدا کردن فایل های مورد نیاز تکیه کنیم.

برای پیدا کردن پوشه ی یک پکیج، از دستور rospack find استفاده کنید:

```
rospack find package-name
```

به طور حتم زمان هایی خواهد بود که شما نام کامل پیکجی که می خواهید را نمی دانید یا به یاد نمی آورید. این مورد بسیار راحت است چون که rospack تکمیل با استفاده از دکمه **tab** را برای اسم پکیج ها پشتیبانی می کند. برای مثال شما می توانید خط زیر را تایپ کنید

```
rospack find turtle
```

و قبل از اینکه دکمه Enter فشار دهید، دکمه ی Tab را دوبار فشار دهید که لیستی از تمام پکیج های نصب شده ی رآس که با turtle شروع می شوند را ببینید. در واقع، بیشتر دستورات رآس این تکمیل شدن با دکمه ی Tab را پشتیبانی می کنند، نه تنها برای نام پکیج، بلکه اکثر جاهایی که این مورد صدق می کند. در دستور بالا، شما می توانید دکمه ی Tab را برای نام دستور rospack و زیر دستور find استفاده کنید.

استفاده متناوب از تکمیل کردن با tab می تواند تعداد چیزهایی را که باید به خاطر بسپارید را کاهش دهد، شامل نام کامل پکیج ها، نودها، تاپیک ها، نوع پیام ها و سرویس ها. کامپیوترها کاملاً در ذخیره سازی و فراخوانی این نوع موارد خوب هستند. توصیه ای ناخواسته: بگذارید کامپیوتر این وظایف را برای شما انجام دهد.

بررسی یک پکیج: برای دیدن فایل های پوشه ی پکیجی، دستور زیر را استفاده کنید:

```
rosls package-name
```

اگر دوست دارید به پوشه ی پکیج بروید، شما می توانید مسیر فعلی را به پوشه ی پکیج مشخصی تغییر دهید، با استفاده از دستورات زیر

roscd package-name

```
1  $ rosls turtlesim
2  cmake
3  images
4  msg
5  package.xml
6  srv
7  $ rosls turtlesim/images
8  box-turtle.png
9  fuerte.png
10         hydro.svg
11         palette.png
12         turtle.png
13         diamondback.png
14         groovy.png
15         indigo.png
16         robot-turtle.png
17         electric.png
18         hydro.png
19         indigo.svg
20         sea-turtle.png
21         $ roscd turtlesim/images/
22         $ eog box-turtle.png
```

لیست (۲-۱) استفاده از rosls و roscd برای دیدن تصاویری که به وسیله ی turtlesim استفاده می شوند. دستور eog نشان دهنده ی تصویر "Eye of Gnome" است.

به عنوان یک مثال ساده، فرض کنید شما می خواهید مجموعه عکس های لاک پشت استفاده شده در turtlesim را ببینید. در لی ست (۲-۱) مثالی برای چگونه ا ستفاده از rosls و roscd برای دیدن لیست این تصاویر و دیدن یکی از آنها می بینید.

در بعضی از نوشته های آن لاین، شما ممکن است مفهومی به نام استک[1] ببینید. یک استک یک مجموعه از پکیج های مرتبط است. که از ویرایش groovy رآس به بعد، مفهوم استک از رده خارج شده است و به وسیله ی متا-پکیج جایگزین شده است.[2,3] بزرگترین تفاوت هم سطح کردن سلسه مراتب است. یک متا-پکیج پکیجی است ـــ که یک بیانیه مانند همه ی پکیج ها دارد، و هیچ پکیج دیگری در پوشه ی آن ذخیره نشده است ـ در حالیکه یک استک شامل پکیج هایی است که در زیر شاخه اش ذخیره شده اند.

۲-۵- مَستِر

تا کنون در مورد فایل ها و چگونه دسته بندی آنها در پکیج ها صحبت کردیم. بیاید تغییر جهت بدهیم و درمورد اینکه چگونه واقعا می توان نرم افزارهای رآس را اجرا صحبت کنیم.
یکی از هدف های اولیه رآس این است که محققین رباتیک بتواند نرم افزاری از مجموعه ی برنامه های کوچک و تقریباً مستقل به نام نُود که به طور همزمان اجرا می شوند را طراحی کند. برای این کار، نودها باید بتوانند با هم ارتباط برقرار کنند. قسمتی از رآس این ارتباط را برقرار می کند که مستر رآس نام دارد. برای شروع مستر دستور زیر را اجرا کنید:

```
roscore
```

ما این حرکت را در مثال turtlesim دیده ایم. برای یکبار، دیگر پیچیدگی دیگری وجود ندارد که نگران آن باشید: roscore هیچ متغیری نمی گیرد و نیاز به هیچ تنظیمی ندارد.
شما باید اجازه دهید که مستر در طول استفاده از رآس اجرا بشود. یک طریقه ی کار منطقی شروع roscore در یک ترمینال است و بعد یک ترمینال دیگر برای کار واقعی استفاده کنید. دلایل زیادی برای متوقف کردن roscore وجود ندارد، به جز وقتی کارتان با رآس تمام می شود. وقتی به این نقطه رسیدید، برای متوقف کردن مستر Ctrl-C را در ترمینالش تایپ کنید.

گرچه نه زیاد، اما مواردی وجود دارد که ریست کردن roscore می تواند ایده ی خوبی باشد. برای مثال، برای عوض کردن یک سری لاگ فایل جدید (در فصل ۴) و یا برای پاک کردن پارامترهای سرویس (در فصل ۷)

[1] http://wiki.ros.org/rosbuild/Stacks

[2] http://wiki.ros.org/catkin/conceptual_overview

[3] http://wiki.ros.org/catkin/package.xml

بیشتر نودهای رآس به مستر متصل می شوند وقتی که شروع می شوند، و برای دوباره متصل کردن آنها تلاش نکنید اگر بعداً ارتباط آنها قطع شد. بنابراین اگر شما roscore را متوقف کنید، تمام نودهای دیگر که در آن زمان اجرا شده اند نمی توانند ارتباط جدیدی را ایجاد کنند، حتی اگر شما roscore را دوباره اجرا کنید.

دستور roscore که در اینجا استفاده شد به وضوح مستر رآس را شروع می کند. در فصل ۶، ما ابزار را یاد خواهیم گرفت که می تواند چند نود را به طور همزمان اجرا کند؛ این ابزار به اندازه ی کافی هو شمند ا ست که اگر م ستری نبود م ستری را شروع کند، اما خو شبختانه م ستر موجود را نیز استفاده خواهد کرد اگر مستری موجود بود.

۲ - ۶ - نودها

وقتی شما roscore را شروع کنید، می توانید برنامه هایی که از رآس استفاده می کنند را اجرا کنید. یک نمونه اجرایی از یک برنامه ی رآس، نود نام دارد. [۱]

عبارت "اجرای نمونه ای از" در این تعریف مهم است. اگر چند کپی از یک برنامه را همزمان اجرا کنیم ـ با توجه به اینکه هر کدام نام نود متفاوتی استفاده می کند ـ با هر کدام از این کپی ها به عنوان یک نود جداگانه رفتار می شود. ما در بخش ۲,۸ این تفاوت را در عمل خواهیم دید.

در م ثال turtlesim ما دو نود ای جاد می کنیم. یکی از نود ها یک نمو نه ی اجرایی به نام turtlesim_node است. این نود مسئول ایجاد پنجره ی turtlesim و شبیه سازی حرکت لاک پشت ا ست. نود دوم یک نمونه اجرایی از turtle_teleop_key است. teleop مختصر شده ی کلمه ی teleoperation است، که به موقعیتی اشاره می کند که در آن انسان رباتی را از راه دور با د ستوری م ستقیم کنترل می کند. این نود برای فشار دادن یک کلید جهت منتظر می ماند، فشار دادن کلید را به دستور حرکتی تبدیل می کند و این دستور را به نود turtlesim_node می فرستد.

شروع نودها: د ستور پایه ای برای ایجاد یک نود (و همچنین شناخته شده به عنوان اجرای یک برنامه ی رآس) rosrun است:[۲]

rosrun package-name executable-name

دو پارامتر برای rosrun مورد نیاز است. اولین پارامتر نام پکیج است. ما در قسمت ۲,۴ روی نام پکیج ها بحث کردیم. پارامتر دوم به صورت ساده نام یک فایل قابل اجرا در این پکیج است.

[۱] http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes

[۲] http://wiki.ros.org/rosbash#rosrun

هیچ جادویی در مورد rosrun وجود ندارد: این فقط یک متن shell است که تشکیلات فایل های رآس را به اندازه کافی می داند که با اســتفاده از اســم پکیج باید کجا دنبال فایل قابل اجرا بگردد. بعد از آنکه فایل را پیدا کرد، rosrun آن را به صـورت عادی اجرا می کند. برای مثال اگر شــما واقعا بخواهید می توانید turtlesim_node را به صــورت مستقیم اجرا کنید، درست مانند همه ی برنامه های دیگر:
/opt/ros/indigo/lib/turtlesim/turtlesim_node
ثبت کردن برنامه به عنوان یک نود رآس در مستر، درون برنامه اتفاق می افتد نه در rosrun.

گوش دادن به نودها : رآس راه هایی برای گرفتن اطلاعات در مورد نودها فراهم کرده است که در زمان مشخصی اجرا می شوند. برای گرفتن لیست نودهای اجرا شده، دستور زیر را اجرا کنید:[1]

```
rosnode list
```

اگر شما این دستور را بعد از اجرای دستور در بخش ۲,۳ اجرا کنید، شما لیست سه نود را خواهید دید:

```
/rosout
/teleop_turtle
/turtlesim
```

چند نکته در مورد لیست بالا قابل ذکر است:

- نود rosout/ یک نود خاص است که با roscore به صورت خودکار شروع می شود. هدفش مانند خروجی استاندارد (std::cout) است که شما ممکن است در برنامه کنسول استفاده کنید. ما به طور کامل در بخش ۴,۴,۲ rosout/ را بررسی می کنیم.

علامت / در rosout/ نشان دهنده ی آن است که این نامِ نود در فضای اسمی اصلی global namespace است. رآس یک سیستم قوی برای اسم نودها و اشیاء دارد. این سیستم، که در فصل ۵ با جزیئات بحث می شود، از فضای اسمی برای سازمان دادن چیزها استفاده می کند.

- دو نود دیگر باید واضــح باشــند: شــبیه ســاز turtlesim و برنامه فرمان دهی از راه دور teleop_turtle که ما در بخش ۲,۳ اجرا کردیم.
- اگر شــما خروجی rosnode list را با نام های فایل های قابل اجرا در دســتور rosrun از بخش ۲,۳ مقایسه کنید، متوجه می شوید که نام نودها ضرورتاً به مانند نام های آن نودها نیستند.

شما می توانید نام نود را صریحاً در دستور rosrun مشخص کنید.

```
rosrun package-name executable-name __name:=node-name
```

[1] http://wiki.ros.org/rosnode

این روش نام نود را که به صورت معمول مشخص شده است را تغییر می دهد. این مورد می تواند مهم باشد چون رآس اصرار دارد که هر نود نام مشخص جداگانه ای داشته باشد. (ما در بخش ۲,۸ برای شکل دادن یک سیستم به عنوان مثالی کمی بزرگتر، از name__ استفاده خواهیم کرد.) به طور کلی، اگر شما از name__ به صورت منظم استفاده می کنید، احتمالاً باید از لانچ فایل استفاده کنید ـ فصل ۶ را مشاهده کنید ـ به جای اینکه نودها را جداگانه اجرا نمایید.

بازرسی نود: برای متوقف کردن یک نود می توانید دستور زیر را اجرا کنید:

```
rosnode kill node-name
```

برخلاف مستر توقف و اجرای دوباره یک نود معمولاً تأثیر زیادی روی دیگر نود نمی گذارد؛ حتی برای نودهایی که با هم پیام رد و بدل می کنند، این ارتباط ها وقتی نودی متوقف می شود قطع می شوند و با ریست کردن نود دوباره منتشر می شوند.

شما می توانید یک نود را با Ctrl-C متوقف کنید. گرچه این روش این شانس را به نود نمی دهد که خودش را از مستر حذف کند. یک نقص توقف کردن نود به این روش این است که نام نود در لیست rosnode list برای مدتی باقی می ماند. این مطلب بی ضرر است، اما ممکن است فهم اینکه چه اتفاقی دارد می افتد را سختتر کند. برای پاک کردن نودهای حذف شده از لیست، می توانید از دستور زیر استفاده کنید:

```
rosnode cleanup
```

۲-۷- تاپیک ها و پیام ها

در مثال turtlesim ما، واضح است که نود فرمان از راه دور و شبیه ساز باید به صورتی با یکدیگر صحبت کنند. در غیر این صورت، چگونه لاک پشت، که در نود دوم فعال است، بفهمد چه موقع در پاسخ به فشار دکمه، که ما در نود اول می خوانیم، حرکت کند؟

یک مکانیزیم ابتدایی که نودهای رآس برای ارتباط استفاده می کنند فرستادن پیام است. پیام ها به عنوان تاپیک در رآس تشکیل شده اند.[۱] ایده این است که یک نود که می خواهد اطلاعاتی را به اشتراک بگذارد پیام ها را در تاپیک یا تاپیک های مناسب منتشر می کند؛ و نودی که می خواهد پیام ها را دریافت کند به تاپیک یا تاپیک های مورد نظرش گوش می کند. مستر رآس مواظب این است که مطمئن شود منتشر کننده و شنونده می توانند همدیگر را پیدا کنند؛ پیام ها مستقیماً از منتشر کننده به شنونده فرستاده می شوند.

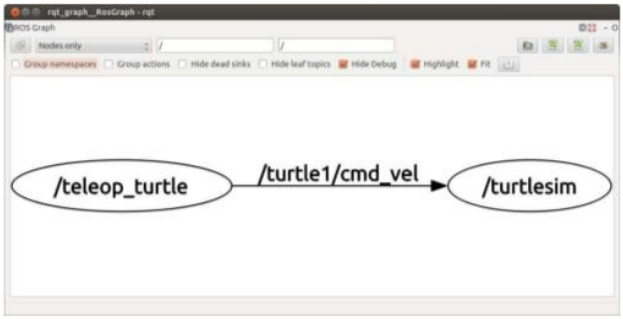
[۱] http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

۲-۷-۱- دیدن گراف

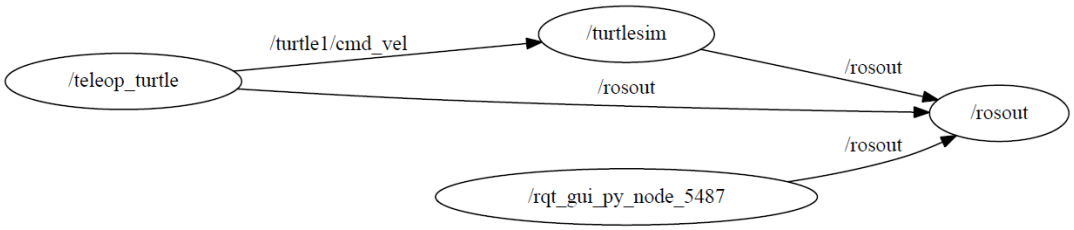
برای مشـاهده ی این ایده (تاپیک ها و پیام ها و از منتشـر کننده به شـنونده) دیدن گراف احتمالاً ساده ترین راه است، و ساده ترین راه برای نمایش دادن ارتباط بین نودهای منتشر کننده/ شنونده رآس استفاده از دستور زیر است:

rqt_graph

در این دســـتور r ارجاع به ROS می دهد و qt به Qt GUI toolkit که برای پیاده ســـازی برنامه استفاده شده است. شما باید یک واسط گرافیکی کاربر (GUI) ببینید، که بیشتر به نمایش دادن نودهای فعال سیستم اختصاص دارد. در این صورت شما شکلی مانند شکل ۲,۲ مشاهده می کنید. در این گراف بیضی ها نشان دهنده ی نودها هستند و جهت بردارها نشان دهنده ی ارتباط بین منتشـــرکننده و شـــنونده اســـت. گراف به ما می گوید که نودی به نام /teleop_turtle پیام /turtle1/cmd_vel را منتشـــر می کند، و نودی به نام /turtlesim/ به پیام ها گوش می کند. (cmd_vel مختصر شده ی "command velocity" به معنی دستور سرعت است.)



شکل (۲-۲) rqt_graph گراف مثال turtlesim را نشان می دهد. نودهای دیباگ مانند rosout به طور پیش فرض حذف شده اند.



شکل (۲-۳) نمودار کامل turtlrsim شامل نودهایی که rqt_graph به عنوان نودهای اشکال زدایی طبقه بندی کرده است.

شما ممکن است متوجه شده باشید که نود rosout که در بخش ۲,۶ دیدیم در این پنجره نیست. به طور پیش فرض، rqt_graph نودهایی که فقط برای ا شکال زدایی ا ست را حذف می کند. شما می توانید این ویژگی را با بردا شتن علامت مربع "Hide debug" غیرفعال کنید. شکل ۲,۳ نتیجه ی گراف را نشان می دهد.

- توجه داشته باشید که **rqt_graph** هم به عنوان یک نود ظاهر شده است.
- همه ی نودها پیام هایشان را روی تاپیکی به نام rosout/ منتشر می کنند، که نود rosout/ به آن گوش دهد. این نود روشی است که نودها می توانند پیام متنی در log لاگ ایجاد کنند. فصل ۴ عمدتا درباره گزارش عملکرد رآس در لاگ است.

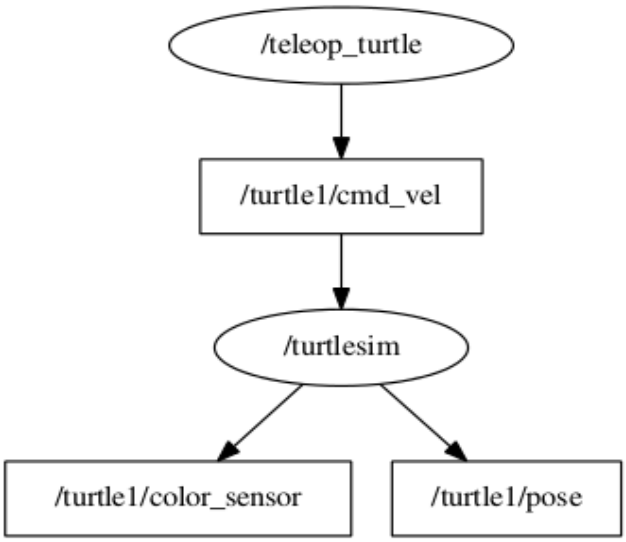
نام rosout/ هم به نود و هم به تاپیک ارجاع می دهد. رآس در مورد این نوع تکرار نام گیج نمی شود چون همیشه از متن مشخص است که ما می خواهیم در مورد نود rosout/ صحبت کنیم یا در مورد تاپیک rosout/ .

این دید از نودهای ا شکال زدایی برای دا شتن ت صویر در ستی از حالت فعلی آنچه می گذرد مفید است، اما همچنین ممکن است گراف را با اطلاعاتی که معمولاً کمکی نمی کنند، پیچیده کند.
ابزار rqt_graph خصیصه هایی دارد که نمایش گراف را تسریع می کند. نویسنده ترجیح می دهد که در فهر ست بالا "Nodes only" را به "Nodes/Topics (all)" تغییر دهد. و همه ی گزینه ها را بجز "Hide Debug" غیرفعال کند. این تنظیمات که نتیجه ی آن در شکل ۲,۴ نشان داده شده است، این مزیت را دارد که همه ی تاپیک ها با مستطیل از نودها جدا شده اند. برای مثال می توانید ببینید که نود turtlesim به علاوه گوش کردن به دستور سرعت، موقعیت فعلی و اطلاعات سنسور رنگ را نیز منتشر می کند. وقتی سیستم جدیدی از رآس را بررسی می کنید، rqt_graph، بخصوص با این ویژگی ها، برای یافتن تاپیک های در دسترس در برنامه برای ایجاد ارتباط با نودهای موجود مفید است.

وجود تاپیک ها بدون شنونده می تواند یک اشکال به نظر برسد اما در واقع این مورد بسیار معمول است. نودهای رآس معمولاً طوری طراحی شده اند که اطلاعات مفیدی که دارند را منتشر می کنند، بدون در نظر گرفتن ایتکه شنونده ای برای آن پیام ها وجود دارد یا نه. این مورد وابستگی بین نودها را کاهش می دهد.

حالا ما می توانیم بخشی از سیستم کنترل از راه دور turtlesim را بفهمیم. وقتی شما کلید را فشار می دهید، نود teleop_turtle/ پیام هایی شامل آن دستورهای حرکتی را با تاپیکی به نام turtle1/cmd_vel/ منتشر می کند. نود turtlesim_node با گوش کردن به آن تاپیک، پیام ها را دریافت می کند، و لاک پشت را با سرعت درخواستی حرکت می دهد. به موارد مهم زیر توجه کنید:

- برای شبیه ساز مهم نیست که کدام برنامه (حتی اگر بداند) cmd_vel را منتشر می کند هر برنامه ای که این تاپیک را منتشر کند می تواند لاک پشت را کنترل کند.
- برای برنامه کنترل از راه دور مهم نیســـت که کدام برنامه به پیام cmd_vel که منتشـــر کرده، گوش می کند. هر برنامه که این تاپیک گوش کند و می تواند آزادانه به آن پاســـخ دهد.



شکل (۲-۴) گراف turtlesim، که همه ی تاپیک ها، شامل تاپیک هایی بدون منتشر کننده یا شنونده، را به عنوان یک شیء جدا نشان می دهد.

به هر حال، اســم این تاپیک ها با turtle1/ شــروع می شــوند چون آنها مربوط به لاک پشــت پیش فرض که اسمش "turtle1"است هستند. ما در فصل ۸ می بینیم که چگونه لاک پشتهای دیگری به پنجره turtlesim اضافه کنیم.

۲-۷-۲- پیام ها و نوع پیام ها

تا کنون ما در مورد ایده هایی که نودها می توانند پیام هایی را به یکدیگر بفرستند صحبت کردیم، اما ما هنوز مشـــخص نکردیم چگونه اطلاعات در پیام ها قرار می گیرند. پس بیایید به پیام ها و تاپیک ها از نزدیکتر نگاه کنیم.

لیست تاپیک ها : برای دیدن فهرست تاپیک ها از دستور زیر استفاده کنید: [1]

```
rostopic list
```

در مثال ما فهرستی از پنج تاپیک می بینید:

```
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

لیســـت تاپیک ها حتما باید مانند مجموعه ی تاپیک های قابل نمایش در rqt_graph باشـــند، اما ممکن است به صورت متن راحتتر باشد.

انعکاس پیام ها: شما می توانید با دستور زیر پیام واقعی منتشر شده روی یک تاپیک مشخص را ببینید:

```
rostopic echo topic-name
```

این دستور تمام پیام های منتشرشده روی تاپیک را روی ترمینال نشان می دهد. لیست ۲,۲ مثال هایی از خروجی دستور زیر را نمایش می دهد:

```
rostopic echo /turtle1/cmd_vel
```

در حالیکه teleop_turtle/ فشـــار کلید را دریافت کرده اســـت. هر خط چین --- در خروجی پایان یک پیام و شروع پیام دیگری را نشان می دهد. در اینجا، سه پیام موجود است.

[1] http://wiki.ros.org/rostopic

```
1  linear :
2  x : 2.0
3  y : 0.0
4  z : 0.0
5  angular :
6  x : 0.0
7  y : 0.0
8  z : 0.0
9  ---
10 linear :
11 x : 0.0
12 y : 0.0
13 z : 0.0
14 angular :
15 x : 0.0
16 y : 0.0
17 z : - 2.0
18 ---
19 linear :
20 x : 2.0
21 y : 0.0
22 z : 0.0
23 angular :
24 x : 0.0
25 y : 0.0
26 z : 0.0
27 ---
```

لیست (۲-۲) مثالی از خروجی rostopic echo

اندازه گیری سرعت انتشار: دو دستور برای اندازه گیری سرعت انتشار پیام و پهنای باند مورد استفاده ی پیام وجود دارد:

rostopic hz topic-name
rostopic bw topic-name

این دستورات به پیام ها گوش می کنند، و در خروجی به ترتیب تعداد پیام در ثانیه و تعداد بایت در ثانیه اندازه می گیرند.

گرچه ممکن است سرعت برای شما اهمیتی نداشته باشد، این دستورات می تواند برای اشکال زدایی مفید باشد، چون این دستورات می توانند به راحتی نشان دهند که پیام ها به صورت متناوب روی تاپیک های مشخصی منتشر شده اند.

بازرسی تاپیک: شما می توانید با دستور rostopic info در مورد تاپیک اطلاعات بیشتری بدست آورید:

rostopic info topic-name

برای مثال از دستور زیر:

rostopic info /turtle1/color_sensor

شما باید خروجی شبیه زیر ببینید:

Type: turtlesim/Color
Publishers:
* /turtlesim (http://donatello:46397/)
Subscribers: None

اولین خط مهمترین قسمت این خروجی است که نوع پیام را نشان می دهد. در مورد /turtle1/color_sensor ، پیام از نوع turtlesim/Color است. کلمه ی "type" در متن ارجاع به مفهوم نوع داده دارد. این بسیار مهم است که نوع پیام را بفهمیم چون می توانیم محتوای پیام را تشخیص دهیم. اینکه نوع پیام یک تاپیک به ما می گوید که هر پیام در آن تاپیک شامل چه اطلاعاتی است، و این اطلاعات چگونه سازمان دهی شده اند.

بازرسی نوع پیام[۲] [۱]: برای دیدن جزیئات نوع پیام از دستور زیر استفاده کنید:

rosmsg show message-type-name

اگر این دستور را برای نوع پیام /turtle1/color_sensor استفاده کنیم:

rosmsg show turtlesim/Color

خروجی این است:

uint8 r
uint8 g
uint8 b

قالب بالا فهرستی از رشته ها (هر خط) است. هر رشته از یک نوع داده (مانند int8 , bool, string) و یک اسم ساخته شده است. خروجی بالا به ما می گوید که turtlesim/Color شامل سه متغییر عددی بدون علامت هشت بیتی (unsigned 8-bit integer) به نام های r , g ,b است. هر پیام از هر تاپیکی با نوع پیام turtlesim/Color به وسیله ی مقادیر این سه رشته تعریف شده است. (همان طور که ممکن هست حدس زده باشید این اعداد به شدت رنگهای قرمز و سبز و آبی برای هر پیکسل در زیر مرکز لاک پشت اختصاص دارد.) .یک مثال دیگر که با آن دوباره به دفعات برخورد خواهیم کرد geometry_msgs/Twist است. این نوع پیام برای تاپیک /turtle1/cmd_vel است، و کمی پیچیده است:

geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x

---

[۱] http://wiki.ros.org/rosms

[۲] http://wiki.ros.org/msg

float64 y
float64 z

در این مورد، هر دوی سرعت خطی linear و زاویه ای angular ر شته های مرکبی از نوع داده ی geometry_msgs/Vector3 هســتند. هر رشــته با نام های x , y , z عضــوی از دو لایه بالاتر هستند. پس هر پیام با نوع geometry_msgs/Twist دارای شش عدد است که در دو بردار به نام های linear و angular ذخیره می شوند. هر عدد از نوع float64 است که طبیعتاً بدین معنی است که هرکدام یک عدد اعشاری ۱۶ بیتی است.

به طور کلی، هر رشته ی مرکب ترکیبی از یک یا چند زیر رشته است که هر کدام ممکن است یک رشــته مرکب دیگر یا یک نوع داده ی ســاده باشــند. این ایده مانند ایده در C++ یا هر زبان برنامه نویسی دیگر است که یک شیء ممکن است دارای عضوهایی از شیء های دیگر باشد.

این مورد ارز شی ندارد که نوع داده های ر شته های مرکب نوع پیام خود شان را دا شته با شند. برای مثال این بسیار منطقی است که تاپیکی با نوع پیام geometry_msgs/Vector3 دا شته باشیم. این نوع پیام شامل سه رشته در لایه بالایی با نام های x , y , z هستند.

این نوع تودرتویی می تواند مفید باشــد، برای جلوگیری از تکرار کد برای ســیســتمی که تعداد زیــادی از نوع پیــام هــایش المــان هــای مشــــترکی دارنــد. یــک مثــال متــداول نوع پیــام std_msgs/Header، که شـــامل توالی برچســـب timestamp و اطلاعات دســـتگاه مختصـــات چارچوب coordinate frame اســت. این نوع به عنوان یک رشــته ی ترکیبی به نام header در صدها نوع پیام دیگر وجود دارد.

خو شبختانه **rosmsg** به صورت خودکار ر شته های مرکب را نیز باز می کند تا نوع ساختار را نشان دهد، با استفاده از فاصله برای نشان دادن ساختار تودرتو، در نتیجه لازم نیست مستقیماً نوع پیام تودرتو را بررسی کنیم.

نوع پیام می تواند همچنین شامل آرایه هایی با طول ثابت یا متغییر (که با براکت [] نشان داده می شوند) و ثابت هایی (به موارد دیگر غیرثابت می گوییم) با شد. این ویژگی برای turtlesim ا ستفاده نشده است. برای مثالی که از این نوع داده استفاده کرده باشد، به sensor_msgs/NavSatFix یک نگاه بندازید، که یک GPS ثابت را نشان می دهد.

منت شر کردن یک پیام با ا ستفاده از نو شتن در ترمینال command line: بیشــتر مواقع انتشار یک پیام با یک برنامه خاص انجام می شود(که در این کتاب یاد می گیریم). اما بعضی اوقات انتشار پیام به صورت دستی ممکن است مفید باشد که با دستور زیر انجام می دهیم :[1]

rostopic pub -r rate-in-hz topic-name message-type message-content

[1] http://wiki.ros.org/rostopic

این د ستور به صورت متناوب پیام داده شده را با تاپیک م شخص شده با سرعت م شخص شده منتشر می کند.

آخرین پارامتر message content باید مقادیر همه ی رشـــته های نوع پیام را به ترتیب مشـــخص کند. به طور مثال:

rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2, 0, 0]' '[0, 0, 0]'

مقادیر برای ر شته های پیام به ترتیبی که rosmsg show ن شان می دهد م شخص می شوند. در این مثال سه عدد اول مربوط به سرعت خطی linear و سه عدد نهایی مربوط به سرعت چرخشی angular است. ما از علامت نقل قول تکی ('. . . ') و براکت ([. . .]) برای تعیین زیر شاخه های گروه های لایه ی بالای رشـــته ی مرکب اســـتفاده کردیم. همان طور که ممکن اســـت حدس زده باشـــید، در این مثال پیام به لاک پشـــت دســـتور می دهد که مســـتقیم (در جهت محور x) بدون چرخش حرکت کند.

با دســـتور مانند زیر به ربات دســـتور می دهیم که حول محور z (عمود بر صـــفحه نمایش شـــما) بچرخد.

rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[0, 0, 0]' '[0, 0, 1]'

در واقع دو رشته ی غیر صفر در دو مثال بالا ـ مشخصاً linear.x و angular.z ـ تنها رشته هایی از geometry_msgs/Twist هســـتند که turtlesim به آنها توجه می کند. چون بقیه ی چهار رشـــته، حرکتی غیر ممکن در شـــبیه ســـاز دو بعد را بیان می کنند، در نتیجه turtlesim آنها را نادیده می گیرد.

د ستور بالا یک ا شکال م شخص دارد که شما باید همه ی زیر ر شته های نوع پیام و همچنین ترتیب آنها را بیاد داشـــته باشـــید. یک روش جایگزین اســـتفاده کردن از یک پارامتر ا ست که همه ر شته ها را به صورت YAML م شخص کرده با شد. (" YAML Ain't Markup Language"[1] زبان نشانه گذاری توسعه پذیر). این دستور (که در واقع شامل کاراکترهای خط جدید است) مانند دستور بالا است، اما مشخصاً نگاشت از اسم به مقدار رشته را نشان می دهد.

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0"
```

[1] http://www.yaml.org/

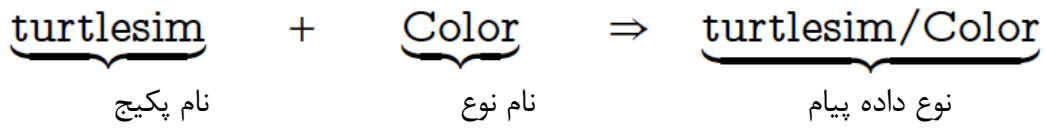
فوت و فن های زیادی برای ارتباط بین bash و YAML وجود دارد که متن های آنلاین کل صفحه را فقط به استفاده از YAML در ترمینال command line اختصاص داده اند.[۲] [۱] ساده ترین راه برای به دست آوردن متن درست استفاده از tab برای تکمیل بقیه ی متن است.
فشار دادن tab بعد از وارد کردن نوع پیام YAML کاملی را نشان می دهد، با تمام همه رشته های نوع داده ی وارد شده. tab پیامی با مقادیر پیش فرض (مانند صفر، رشته ی خالی، false و ...) ایجاد می کند، اما شما می توانید مقادیر را به مقادیر مورد نظرتان تغییر دهید.

گزینه های دیگری برای استفاده از rostopic pub وجود دارد:

- در حالت نشان داده شده r- برای انتخاب مد سرعت در rostopic pub استفاده شده است، که پیام را در تناوب مشخصی منتشر می کند. این دستور همچنین مد یکبار (-۱ " dash one") را نیز پشتیبانی می کند و همچنین مد خاصی با قفل (-۱ "dash ell") که فقط یکبار منتشر می کند اما مطمئن خواهد شد که شنونده ی جدید آن تاپیک، پیام را دریافت کرده است. مد قفل دار در واقع مد پیش فرض است.
- همچنین این امکان وجود دارد که پیام را از فایلی با استفاده از f- یا از یک ورودی استاندار (با حذف f- و محتوی پیام از دستور) بخوانیم. در هر دو مورد ورودی باید مانند خروجی rostopic echo باشد.

ممکن است شما شروع به تصور حالت های ممکنی که با استفاده از ترکیب دستورهای rostopic echo و rostopic pub به عنوان راهی برای ضبط و نشر مجدد پیام کرده باشید، برای تست کردن خودکار برنامه تان. در این صورت، ابزار rosbag (در فصل ۹) باید برایتان جالب باشد، که پیاده سازی کاملی از این ایده است.

فهمیدن ا سم نوع پیامها : مانند همه چیز در رآس، هر نوع پیام به پکیج مشخصی تعلق دارد. اسم نوع پیام همیشه شامل اسلش / است و قسمت قبل از اسلش نام پکیجش است:
package-name/type-name
برای مثال turtlesim/Color به دو بخش تقسیم می شود:



[۱] http://wiki.ros.org/YAMLOverview

[۲] http://wiki.ros.org/ROS/YAMLCommandLine

این تقسیم اسم نوع پیام چند هدف دارد:

- مهمترین هدف، جلوگیری از تداخل اسم ها با استفاده از به کار بردن نام پکیج در اسم نوع پیام است. برای مثال دو نوع پیام geometry_msgs/Pose و turtlesim/Pose مشخصاً دو نوع پیام متفاوت هستند که دارای داده های مختلف هستند (البته با مفهوم مشابه).
- همان طور که در فصل ۳ می بینیم، وقتی برنامه ای در رآس می نویسیم، ما باید وابستگی ها به پکیج های دیگر را مشخص کنیم که شامل نوع پیام های مورد استفاده نیز می شود. وجود نام پکیج به عنوان قسمتی از نوع پیام، پیدا کردن این وابستگی ها را ساده تر می کند.
- در آخر، دانستن پکیجِ نوع داده، برای دریافتن هدفش کمک می کند. برای مثال، اسم نوع ModelState به تنهایی گنگ است، اما اسم کامل gazebo/ModelState مشخص می کند که نوع داده قسمتی از شبیه ساز Gazebo است، و شامل اطلاعاتی در مورد یکی از مدلهای شبیه ساز است.

۲-۸- یک مثال بزرگ

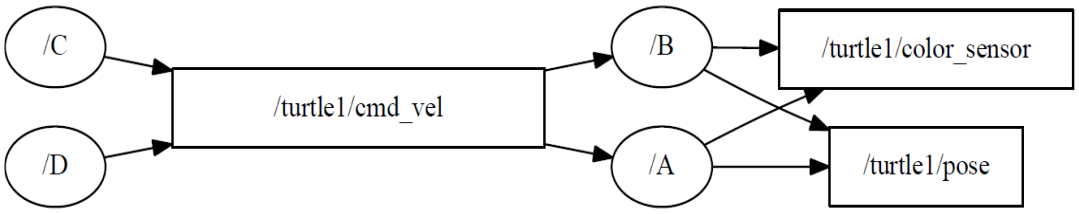
تا اینجا در این فصل، ما یاد گرفتیم چگونه مستر رآس را اجرا کنیم، چگونه نودهای رآس را شروع کنیم، و چگونه تاپیک هایی که نودها برای ارتباط با یکدیگر ا ستفاده می کنند را برر سی کنیم. در این قسمت کار را با مثالی کمی بزرگتر از مثال بخش ۲,۳ ادامه می دهیم، تا طرز کار پیام ها و تاپیک ها را کمی بیشتر روشن کنیم.

ابتدا، تمام نودهای فعال را متوقف کنید. roscore را اگر فعال نیست اجرا کنید. بعد در چهار ترمینال جداگانه چهار دستور زیر را اجرا کنید:

```
rosrun turtlesim turtlesim_node __name:=A
rosrun turtlesim turtlesim_node __name:=B
rosrun turtlesim turtle_teleop_key __name:=C
rosrun turtlesim turtle_teleop_key __name:=D
```

این کار باید دو نمونه از شبیه ساز لاک پشت turtlesim را ـ در دو پنجره ی جداگانه ـ و دو نمونه از نود کنترل از راه دور turtlesim شروع کند.

تنها عنصری که ممکن است در rosrun ناآشنا باشد پارامتر name__ است.این پارامتر اسم پیش فرض نود را تغییر می دهد. به این پارامترها نیاز داریم چون مستر رآس اجازه نمی دهد چند نود اسم مشابه داشته باشند.



شکل (۲-۵) گرافی پیچیده از رآس، با دو نود turtlesim به نام های A و B و نودهای کنترل از راه دور به نام های C و D.

- اگر شما تلاش دارید دو نود را با اسم مشابه شروع کنید، نود جدید بدون مشکل شروع می شود، اما نود اولیه با پیام مشابه زیر متوقف می شود:

[WARN] [1369835799.391679597]: Shutdown request received.
[WARN] [1369835799.391880002]: Reason given for shutdown:
[new node registered with same name]

گرچه ما در اینجا از این کار پرهیز کردیم اما ممکن است این کار مفید باشد. بخصوص وقتی شما دارید اشکال زدایی می کنید و دوباره نود را اجرا می کنید، چون این کار باعث می شود شما مطمئن باشید که چند ویرایش از یک نود اشتباهاً با هم اجرا نشده اند.

قبل از اینکه ما در مورد این چهار نود بحث کنیم، شاید شما مایل باشید در مورد نحوه ی کارکرد سیستم کمی فکر کنید. گراف که با rqt_graph نمایش داده می شود چگونه خواهد بود؟ کدام لاک پشت در پاسخ به کدام فرمان از راه دور حرکت خواهد کرد؟

امیدواریم که پیش بینی شما به مانند شکل ۲,۵ باشد، و هر دو لاک پشت در پاسخ به فشار دادن کلید در هریک از دو نود فرمان از راه دور حرکتی مشابه خواهند داشت. حالا ببینیم چرا.

۲-۸-۱- ارتباط به وسیله ی تاپیک چند به چند است

شما شاید انتظار داشتید که هر نود فرمان از راه دور به یک شبیه ساز متصل شود، و دو شبیه سازی قابل کنترل مجزا داشته باشید.(۳.در فصل ۶ روش درست برای ایجاد دو شبیه ساز مستقل به موازات هم را یاد می گیریم.) توجه داشته باشید که گرچه این دو نوع نود به ترتیب تاپیک /turtle1/cmd_vel را منتشر می کنند و گوش می کنند. پیامی که در این تاپیک منتشر می شود، بدون در نظر گرفتن به اینکه کدام نود آن را منتشر می کند، به وسیله هر نودی که به این تاپیک گوش می کند قابل دریافت است.

در این مثال، هر پیامی که با نود فرمان از راه دور C منتشـر می شـود برای هر دو نود A و B قابل دریافت است. همان طور پیام منتشر شده به وسیله ی نود D برای هر دو نود A و B قابل دریافت است. وقتی پیام دریافت شود لاک پشت بدون در نظر گرفتن نود ناشر متناسب حرکت می کند. ایده ی اصلی اینجا استفاده از پیام برای ارتباط خیلی-به-خیلی است. ناشرها و شنونده های زیادی می توانند یک تاپیک را به اشتراک بگذارند.

۲-۸-۲- نودها خیلی کم به هم وابسته اند.

شکی نیست که شما متوجه شده اید که ما نیاز نداریم که شبیه ساز turtlesim را بازنویسی کنیم که دستور حرکت را از منابع (نودهای) مختلف بپذیرید. همچنین نیازی نیست نود فرمان از راه دور طوری طراحی شـود که چند نمونه از شـبیه سـاز را همزمان کنترل کند. در واقع بسـط دادن این مثال به تعداد زیادی از این نودها تمرین ساده ای است. (به همین دلیل در کامپیوتر نویسنده، بعد از شروع ۱۰۰ نمونه شبیه ساز turtlesim_node، به داشتن تعداد زیاد active x client اعتراض کرد!)

یک شرایط حاد دیگر را در نظر بگیریم، چه اتفاقی خواهد افتاد اگر شبیه ساز turtlesim به تنهایی بدون هیچ نود دیگری شروع شود؟ در این شرایط، شبیه ساز منتظر پیام های /turtle1/cmd_vel خواهد بود، گرچه این موضوع واضح است که ناشری برای آن تاپیک وجود ندارد.

مطلب اساسی این است که نود turtlesim ـــ و بیشتر نودهای رآس که خوب طراحی شده باشند- وابسـتگی کمی به هم دارند. هیچ کدام از نودها دقیقاً در مورد هیچ کدام از نودهای دیگر نمی دانند؛ ارتباط آنها فقط غیرمستقیم است، و در سطح تاپیک ها و پیام هاست. این استقلال نودها، همراه با شـکسـتن خدمات یک وظیفه ی بزرگ به قسـمت های کوچک، از ویژگی های کلیدی طراحی رآس است.

- نرم افزارها (مانند turtle_teleop_key) که اطلاعات ایجاد می کنند، می توانند این اطلاعات را منتشر کنند، بدون در نظر گرفتن اینکه چگونه استفاده خواهند شد.
- نرم افزارهایی (مانند turtlesim_node) که از اطلاعات اسـتفاده می کنند می توانند به تاپیک ها و داده های مورد نیازشـان از تاپیک گوش کنند، بدون در نظر گرفتن اینکه این اطلاعات چگونه ایجاد شده اند.

رآس شیوه ای به نام سرویس دارد که بیشتر مستقیم و ارتباط یک-به-یک است. این تکنیک دوم کمتر متداول اسـت، اما موارد اسـتفاده ی خودش را دارد. فصـل ۸ چگونگی ایجاد و اسـتفاده از سرویس را توضیح می دهد.

۲-۹- چک کردن اشکالات

یک دستور نهایی (فعلاً) که می تواند کمک کننده باشد وقتی که رآس آن گونه که انتظار داریم رفتار نمی کند، دستور roswtf است.[۲] [۱](اختصار این کلمه در اسناد ذکر نشده است اما نویسنده کاملاً مطمئن است که این دستور اختصار جمله ی ?Why The Failure به معنی چرا خطا؟ است.) که بدون عنصر اضافه قابل اجرا است:

```
roswtf
```

این دستور چکاپ های متنوعی را شامل تست متغییرهای محیطی، نصب فایل ها، اجرای نودها اجرا می کند. برای مثال roswtf چک می کند که آیا روند نصب **rosdep** تکمیل شده است یا نه، آیا هیچ نودی یک دفعه هنگ یا غیرفعال شده است یا نه، آیا نودهای فعال به درستی به هم مرتبط شده اند یا نه. فهرست کاملی از چکاپ های roswtf متأسفانه فقط به صورت کد اصلی پایتان موجود است.

۲-۱۰- در ادامه خواهیم دید

هدف اصلی این بخش معرفی بخش های پایه ای رآس مانند پیام ها، تاپیکها، همراه با بعضی ابزارهای دستوری در ترمینال برای ایجاد ارتباط بین بخش ها بود. در فصل بعدی، ما فراتر از ایجاد ارتباط بین برنامه های موجود در رآس می رویم و برنامه ی جدیدی را می نویسیم.

---

[۱] http://wiki.ros.org/roswtf

[۲] http://wiki.ros.org/ROS/Tutorials/Gettingstartedwithroswtf

فـصل ۳ : نوشتن برنامه در رآس

در این فصل ما برنامه ی منتشر کردن و گوش دادن به پیام در رآس را می نویسیم:

تا اینجا ما چند ویژگی ا صلی رآس را معرفی کردیم، شامل پکیج ها، نودها، تاپیک ها، و پیام ها. ما همچنین زمانی را برای بررسی برنامه های موجود ساخته شده بر اساس این ویژگی ها صرف کردیم. حالا زمان این است که برنامه ی خودمان را در رآس بنویسیم. در این فصل یاد می گیریم چگونه فضای کاری را تنظیم کنیم و سه برنامه ی کوتاه شامل مثال استاندارد "hello world" و دو برنامه ی چگونه منتشر کردن و گوش کردن به پیام ها را نشان می دهیم.

۳-۱- ساختن یک فضای کاری و یک پکیج

ما در بخش ۲,۴ دیدیم که همه ی برنامه های رآس، شامل برنامه ای که ما ایجاد خواهیم کرد، به صورت یک پکیج سازماندهی می شود. قبل از اینکه هر برنامه ای بنویسیم، قدم اول ایجاد یک فضای کاری برای نگهداری پکیج هایمان است، و قدم بعدی ساختن خود پکیج است.

ساختن فضای کاری: پکیجی که شما ایجاد می کنید باید در پو شه ای به نام فضای کاری قرار بگیرد.[۱] برای مثال، فضای کاری نویسنده در پوشه ای به نام home/jokane/ros/ است. اما شما می توانید اسم فضای کاری خودتان را هرچه دوست دارید بگذارید، و در هر مسیری که ترجیح می دهید آن را ذخیره کنید. به صورت معمول می توان از دستور mkdir برای ایجاد یک پوشه استفاده کرد. ما به این پوشه، پوشه ی فضای کاری خواهیم گفت.

> برای کابران بسیاری، بیش از یک فضای کاری واقعاً نیاز نیست. گرچه سیستم کمپایل کتکین catkin در رآس، که ما در بخش ۳,۲,۲ معرفی خواهیم کرد، همه ی پکیج های موجود در یک فضای کاری را همزمان کمپایل می کند. بنابراین، اگر شما روی پکیج های بسیاری کار می کنید و پروژه های مختلف جداگانه ای دارید، ممکن است استفاده از چند فضای کاری مستقل مفید باشد.

قدم نهایی تنظیم فضای کاری است. یک زیرپوشه (زیرشاخه) به نام src درون پوشه ی فضای کاری ایجاد کنید. همان طور که شاید حدس زده باشید کد اصلی پکیج های شما در این پوشه قرار خواهد گرفت.

ساختن یک پکیج د ستور: برای ایجاد یک پکیج رآس، که شما باید در پوشه ی src فضای کاریتان اجرا کنید، به صورت زیر است: [۲]

[۱] http://wiki.ros.org/catkin/Tutorials/create_a_workspace

[۲] http://wiki.ros.org/ROS/Tutorials/CreatingPackage

catkin_create_pkg package-name

در واقع این دستور کار زیادی انجام نمی دهد: یک پوشه برای پکیج ایجاد می کند و دو فایل تنظیمات درون پوشه ایجاد می کند.

- در فایل اول تنظیمات به نام package.xml شامل متنی که در بخش ۲,۴ در موردش بحث کردیم است.
- دومین فایل به نام CMakeLists.txt، که دستورات متنی برای سیستم صنعتیِ ساختن چند-سخت افزاری (کراس پلت فورم) به نام CMake است. در واقع لیستی از ساختار ساختن پکیج است: چه فایل اجرایی باید ساخته شود، چه کدی برای ساختن هر کدام استفاده شود، کجا می توان فایل های include و کتابخانه ای را برای آنها پیدا کرد. CMake به صورت داخلی در کتکین catkin استفاده می شود.

در بخش بعدی، تغییراتی که در هر یک از فایل ها برای تنظیم پکیج جدیدتان باید بدهید را خواهیم دید. برای الآن همین کافی هست که بدانید دستور catkin_create_pkg کار خارق العاده ای انجام نمی دهد. وظیفه ی آن به سادگی ایجاد پوشه ی پکیج و دو فایل تنظیمات با متن پیش فرض برای راحتی شماست.

این ساختار سه لایه پوشه ها، پوشه ی فضای کاری، پوشه ی src، پوشه ی شامل پکیج ـ ممکن است برای ساختن یک پکیج ساده و یک فضای کاری کوچک بی خود به نظر برسد، اما سیستم ساختن catkin به آن نیاز دارد.

اسم پکیج ها در رآس قردادی را دنبال می کند که فقط حروف کوچک، اعداد، خط تیره را اجازه می دهد. طبق قرارداد اول اسم باید یک حرف کوچک باشد. و بعضی از ابزار رآس، شامل catkin، با دنبال نکردن این قرارداد مشکل دارند.

همه ی مثال های این کتاب به پکیجی به نام agitr تعلق دارند، که اول کلمات اسم کتاب به انگلیسی است. اگر شما می خواهید این پکیج را دوباره ایجاد کنید، می توانید با اجرای دستور زیر در پوشه ی src فضای کاریتان پکیجی با این نام ایجاد کنید:

catkin_create_pkg agitr

یک روش دیگر دانلود کردن پکیج از وب سایت کتاب است، و قرار دادن آن در پوشه ی فضای کاری است.

تغییر متن تو صیفات (مانیفست): بعد از ایجاد پکیج شاید شما مایل با شید package.xml را تغییر دهید، که شامل متا-داده برای توصیف پکیج است. در ویرایش پیش فرض نصب شده با

catkin_create_pkg به صورت دلخواه دارای تو ضیحاتی زیادی ا ست. توجه دا شته با شید گرچه رآس بیشتر این توضیحات را استفاده نمی کند، چه در زمان ساختن چه در زمان اجرا، اگر بخواهید آنها را در دسترس عموم بگذارید بسیار مهم هستند. برای هماهنگ کردن توصیفات با کاربرد واقعی پکیج، حداقل قسمت توصیفات (description) و نگهدارنده (maintainer) باید پر شوند. در لیست ۳,۱ متن توصیفات پکیج agitr آورده شده است.

```
1  <?xml version="1.0"?>
2  <package>
3  <name>agitr</name>
4  <version>0.0.1</version>
5  <description>
6     Examples from A Gentle Introduction to ROS
7  </description>
8  <maintainer email="jokane@cse.sc.edu">
9  Jason O' Kane
10 </maintainer>
11 <license>TODO</license>
12 <buildtool_depend>catkin</buildtool_depend>
13 <build_depend>geometry_msgs</build_depend>
14 <run_depend>geometry_msgs</run_depend>
15 <build_depend>turtlesim</build_depend>
16 <run_depend>turtlesim</run_depend>
17 </package>
```

لیست (۳-۱) متن توصیفی (package.xml) برای پکیج agitr این کتاب

۳-۲- سلام رآس!

حالا که یک پکیج درست کردیم، می توانیم در رآس برنامه بنویسیم. یک برنامه ساده

لیست ۳,۲ ویرایشی از برنامه ی استاندارد "Hello, world!" در رآس را نشان می دهد. فایل اصلی hello.cpp نام دارد که در پوشـــه پکیج شـــما، دقیقاً در کنار package.xml و CMakeLists.txt است.

```
1  // This is a ROS version of the standard "hello, world"
2  // program.
3
4  // This header defines the standard ROS classes.
5  #include <ros/ros.h>
6
7  int main(int argc, char **argv) {
8    // Initialize the ROS system.
9    ros::init(argc, argv, "hello_ros");
10
11   // Establish this program as a ROS node.
12   ros::NodeHandle nh;
13
14   // Send some output as a log message.
15   ROS_INFO_STREAM("Hello, ROS!");
16 }
```

لیست (۳-۲) نمونه از برنامه ای به نام hello.cpp در رآس

بعضی دستورکارهای آنلاین توصیه می کنند یک پوشه به نام src در پوشه ی پکیج خود درست کنید که شامل فایل های اصلی C++ باشد. این سازماندهی می تواند مفید باشد بخصوص برای پکیج های بزرگتر با انواع فایل های مختلف، اما این کار حتماً الزامی نیست.

ما چند لحظه ی دیگر چگونه کمپایل و اجرا کردن این برنامه را تو ضیح خواهیم داد اما ابتدا بیایید خود کد را بررسی کنیم.

- اولین (header file) سربرگ ros/ros.h شامل کلاس های استاندارد رآس است. شما آن را در تمام برنامه هایی که می نویسید استفاده خواهید کرد.
- تابع ros::init کتابخانه ی مورد استفاده ی رآس را مقداردهی اولیه می کند. آن را یکبار در اول برنامه ی خود فرابخوانید. [۱] آخرین پارامتر یک رشته شامل نام پیش فرض نود شماست.

این نام پیش فرض می تواند به وسیله فایل launch (صفحه ی ۸۶) یا با پارامتر دستور rosrun(صفحه ی ۲۰) تغییر کند.

[۱] http://wiki.ros.org/roscpp/Overview/InitializationandShutdown

- شی (ros::NodeHandle (object اصلی ترین مکانیزمی است که برنامه ی شما برای ارتباط با سیستم رآس استفاده می کند.[1] ساختن این شی برنامه ی شما را به عنوان یک نود در مستر رآس ثبت می کند. ساده ترین تکنیک ایجاد یک شی NodeHandle تنها برای استفاده در سراسر برنامه است.

به صورت داخلی، کلاس NodeHandle دارای شمارنده ی مرجعی است که نود جدید را در مستر ثبت می کند، وقتی اولین شی NodeHandle ساخته شد. همچنین وقتی نود حذف می شود که همه ی شی های NodeHandle از بین رفته باشند. این جزیئات در دو مورد مؤثرند: اولا شما می توانید، اگر ترجیح بدهید، چند NodeHandle درست کنید، در حالیکه همه ی آنها به یک نود مربوط اند. مواردی خواهد بود که این کار لازم است. یک مثال از این مورد در صفحه ی ۱۲۷ آورده شده است. ثانیا این بدین معنی است که استفاده از ارتباط دهنده ی استاندارد roscpp برای اجرای چند نود مستقل در یک برنامه غیرممکن است.

- خط ROS_INFO_STREAM پیام اطلاعاتی را ایجاد می کند. این پیام به جاهای مختلفی فرستاده می شود، شامل صفحه ی کنسول. ما جزیئات بیشتری در این مورد در فصل ۴ یاد خواهیم گرفت.

۳-۲-۱- کمپایل کردن برنامه Hello

چگونه یک برنامه را کمپایل و اجرا کنیم؟ این کار با سیستم ساختن رآس به نام کتکین catkin انجام می شود. که شامل چهار مرحله است.[2]

مشخص کردن وابستگی ها اول شما باید پکیج های دیگری که ما به آنها وابسته ایم را مشخص کنید. برای برنامه ی C++، ابتدا در این مرحله نیاز است که مطمئن شویم catkin کمپایلر C++ را فراهم کرده است، کامپایلر C++ با پرچم های (flag) مناسب که برای قرار دادن سربرگ ها و کتابخانه های نیاز دارد.

برای لیست کردن وابستگی ها CMakeLists.txt را در پوشه ی پکیجتان تغییر دهید. ویرایش پیش فرض خط زیر را دارد:

(catkin REQUIRED) find_package

[1] http://wiki.ros.org/roscpp/Overview/NodeHandles

[2] http://wiki.ros.org/ROS/Tutorials/BuildingPackages

وابستگی به دیگر پکیج های کتکین می تواند در قسمت COMPONENTS این خط، اضافه شود:

(catkin REQUIRED COMPONENTS package-names) find_package

برای مثال hello، ما به وابستگی به نام roscpp نیاز داریم، که کتابخانه های C++ مورد استفاده ی رآس را فراهم می کند. بنابراین خط مورد نیاز find_package اینگونه است:

(catkin REQUIRED COMPONENTS roscpp) find_package

ما باید لیست وابستگی ها را در package.xml پکیج بیاوریم، با استفاده از المان های build_depend و run_depend:

```
<build_depend>package-name</build_depend>
<run_depend>package-name</run_depend>
```

در مثال ما، برنامه ی hello به roscpp هم در زمان ساختن و هم در زمان اجرا نیاز دارد پس مانیفست باید شامل هر دو باشد:

```
<build_depend>roscpp</build_depend>
<run_depend>roscpp</run_depend>
```

گرچه، وابستگی های مشخص شده در مانیفست در پروسه ی ساختن(build) استفاده نمی شود؛ اگر شما آن را اینجا حذف کنید، احتمالاً شما هیچ خطایی دریافت نمی کنید تا زمانی که پکیجتان را پخش کنید و کسی بدون داشتن پکیج های مورد نیاز سعی به ساختن آن کند.

م شخص کردن فایل قابل اجرا: در قدم بعد، ما باید دو خط به CMakeLists.txt ا ضافه کنیم تا فایل قابل اجرای مورد نیاز را ایجاد کنیم. حالت کلی این است:

```
add_executable(executable-name source-files)
target_link_libraries(executable-name ${catkin_LIBRARIES})
```

در خط اول نام فایلِ قابل اجرا و همچنین لیست سورس کدهایی که باید برای ایجاد آن باید کمپایل بشوند را مشخص می کنیم. اگر شما بیش از یک سورس کد داشته باشید، همه ی آنها را لیست کنید، و با استفاده از فاصله آنها را از هم جدا کنید. در دومین خط به CMake می گوید که وقتی از این فایلِ قابل اجرا استفاده می کند کدام پرچم کتابخانه ی مناسب را استفاده کند (م شخص شده در خط find_package در بالا). اگر پکیج شما شامل بیش از یک فایلِ قابل اجرا باشد، این دو خط را برای هر فایلِ قابل اجرا کپی و اصلاح کنید. در مثال ما، می خواهیم فایلِ قابل اجرایی به نام hello از سورس کد hello.cpp را ایجاد نماییم، بنابراین ما باید دو خط زیر را به CMakeLists.txt اضافه کنیم:

```
add_executable(hello hello.cpp)
target_link_libraries(hello ${catkin_LIBRARIES})
```

```
1  # What version of CMake is needed?
2  cmake_minimum_required(VERSION 2.8.3)
3
4  # The name of this package.
5  project(agitr)
6
7  # Find the catkin build system, and any other packages on which we
8  depend.
9  find_package(catkin REQUIRED COMPONENTS roscpp)
10
11 # Declare our catkin package.
12 catkin_package()
13
14 # Specify locations of header files.
15 include_directories(include ${catkin_INCLUDE_DIRS})
16 # Declare the executable, along with its source files.If
17 # there are multiple executables , use multiple copies of
18 # this line
19 add_executable(hello hello.cpp)
20 # Specify libraries against which to link.Again, this
21 # line should be copied for each distinct executable in
22 # the package.
23 target_link_libraries(hello ${catkin_LIBRARIES})
24
```

لیست (۳-۳) یک CMakeLists.txt برای ساختن hello.cpp

بعنوان مرجع، لیست ۳,۳ یک CMakeLists.txt کوتاه که برای مثال ما کافی است را نشان می دهد. CMakeLists.txt پیش فرض تولید شده به وسیله ی catkin_create_pkg شامل راهنماهای غیرفعال شده ی برای اهداف دیگر است؛ برای بسیاری از برنامه های ساده، چیزی شبیه این ویرایش ساده که اینجا نشان داده شده است کافی است.

ساختن ف ضای کاری: وقتی CMakeLists.txt را تنظیم کردید، شما می توانید فضای کاری خودتان را بسازید (شامل کمپایل همه ی فایل های قابل اجرای پکیجیتتان) با ا ستفاده از د ستور زیر:

catkin_make

چون این دستور طوری طراحی شده است که همه ی پکیج های محیط کارتان را کمپایل می کند، این دستور باید از پوشه ی فضای کاری اجرا شود. این دستور مراحل تنظیمات مختلفی را اجرا می کند (بخصوص بار اول که اجراش می کنید) و دو زیر پوشه به نام های devel و build در محیط کارتان ایجاد می کند. این دو زیر پو شه شامل فایل های وابسته به ساخت مانند makefile هایی

که به صورت خودکار ایجاد می شوند، شی کد، و فایل های قابل اجرا هستند. اگر شما دو ست داشته باشید می توانید بعد اتمام کارتان با خیال راحت زیر پوشه های devel و build را حذف کنید.

اگر خطای حین کمپایل وجود داشته باشد در این مرحله می بینید و بعد از درست کردن خطاها، می توانید catkin_make را برای تکمیل روند ساخت دوباره اجرا کنید.

اگر شما خطایی از catkin_make مبنی بر اینکه نمی تواند ros/ros.h را پیدا کند، یا مراجع "undefined reference" را در ros::init یا هر تابع دیگری پیدا نمی کند، بیشتر اوقات دلیلش این است که وابستگی roscpp در CMakeLists.txt آورده نشده است.

مشخص کردن setup.bash به عنوان منبع: آخرین مرحله، اجرای فایل قابل اجرای setup.bash است، که به وسیله ی catkin_make در داخل زیرپوشه ی devel در فضای کارتان ساخته شده است:

```
source devel/setup.bash
```

این کار به صورت خودکار متن قابل اجرایی (اسکریپتی) را ایجاد می کند که متغییرهای محیطی مختلفی را تنظیم می کند تا رآس بتواند پکیج جدید شما و فایل های قابل اجرایش را پیدا کند. این مورد مشابه setup.bash کلی در بخش ۲,۲ است، اما مشخصاً برای فضای کاری شما. تا زمانی که ساختار فضای کارتان تغییر نکند، شما فقط یکبار این کار را برای هر ترمینال باید انجام دهید، حتی اگر کدتان را اصلاح و دوباره با استفاده از catkin_make کمپایل کنید.

۳-۲-۲- اجرای برنامه ی hello

وقتی همه ی این مراحل کامل شد، برنامه جدید شما در رآس با استفاده از rosrun قابل اجرا است (بخش ۲,۶) ، درست مانند هر برنامه ی دیگری از رآس. برای این مثال دستور زیر را اجرا کنید:

```
rosrun agitr hello
```

این برنامه باید خروجی مانند زیر ایجاد کند:

```
[ INFO] [1416432122.659693753]: Hello, ROS!
```

فراموش نکنید ابتدا باید roscore را شروع کنید: این برنامه یک نود است و هر نود برای درست اجرا شدن به مستر نیاز دارد. به هر حال، عدد در خط خروجی زمان اندازه گیری شده به ثانیه از اول ژانویه ۱۹۷۰ ـ تا زمان اجرای خط ROS_INFO_STREAM برنامه ی ما را نشان می دهد.

rosrun همراه با دستورات دیگری از رآس ممکن است خطایی مشابه زیر ایجاد کند:
[rospack] Error: stack/package package-name not found
دو دلیل رایج این خطا نوشتن غلط اسم پکیج و درست اجرا نکردن فایل setup.bash برای فضای کارتان است.

۳-۳- برنامه ی منتشر کننده ی پیام

برنامه ی hello در قسمت قبل نشان داد که چگونه یک برنامه ی ساده را کمپایل و اجرا کنیم. این برنامه برای معرفی کتکین مفید بود، اما همه برنامه های "Hello,World!" کار خاصی انجام نمی دهد. در این بخش، ما به برنامه ی که به رآس بیشتر مربوط است نگاهی می اندازیم.[۱] بخصوص، ما خواهیم دید که چگونه سرعتی که به صورت تصادفی ایجاد شده است را به لاک پشت turtlesim بفرستیم، که باعث می شود بدون هدفی حرکت کند. خلاصه ی کد C++ برنامه به نام pubvel را در لیست ۳,۴ ببینید. این برنامه همه ی اجزای مورد نیاز برای منتشر کردن پیام را در کد نشان می دهد.

```
1  //This program publishes randomly-generated velocity messages for turtlesim.
2  #include <ros/ros.h>
3  #include <geometry_msgs/Twist.h>  // For geometry_msgs::Twist
4  #include <stdlib.h> // For rand() and RAND_MAX
5  int main(int argc, char **argv) {
6    // Initialize the ROS system and become a node.
7    ros::init(argc, argv, "publish_velocity");
8    ros::NodeHandle nh;
9    // Create a publisher object.
10   ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
11     "turtle1/cmd_vel", 1000);
12   // Seed the random number generator.
13   srand(time(0));
14   // Loop at 2Hz until the node is shut down.
15   ros::Rate rate(2);
16   while(ros::ok()) {
17     // Create and fill in the message.  The other four
```

[۱] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(C++)

```
18    // fields, which are ignored by turtlesim, default to 0.
19    geometry_msgs::Twist msg;
20    msg.linear.x = double(rand())/double(RAND_MAX);
21    msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
22    // Publish the message.
23    pub.publish(msg);
24    // Send a message to rosout with the details.
25    ROS_INFO_STREAM("Sending random velocity command:"
26      << " linear=" << msg.linear.x
27      << " angular=" << msg.angular.z);
28    // Wait until it's time for another iteration.
29    rate.sleep();
30  }
31 }
```

لیست (۳-۴) برنامه ای به نام pubvel.cpp که عددی به صورت تصادفی برای لاک پشت turtlesim منتشر می کند

۳-۳-۱- منتشر کردن پیام

مهمترین تفاوت بین pubvel و hello ریشه در منتشر کردن پیام دارد.

اضافه کردن فایل نوع پیام

شما ممکن است از بخش ۲,۷,۲ به یاد بیاورید که هر تاپیکِ رآس مربوط به یک نوع پیام است. هر نوع پیام یک فایل سربرگ در C++ دارد. شما باید با استفاده از include# این سربرگ را برای هر نوع پیام موجود در برنامه اضافه کنید، با استفاده از کدی مانند زیر:

```
#include <package_name/type_name.h>
```

توجه داشته باشید که نام پکیج در اینجا باید نام پکیج شامل نوعِ پیام باشد و نه لزوما نام پکیج شما. در pubvel، ما می خواهیم پیامی از نوع geometry_msgs/Twist ـــــ پیامی از نوع Twist متعلق به پکیجی به نام geometry_msgs ـ منتشر کنیم پس به خط زیر نیاز داریم:

```
#include <geometry_msgs/Twist.h>
```

هدف از این سربرگ تعریف کردن کلا سی از C++ ا ست که دارای ع ضوی به مانند ر شته ی نوعِ پیام مورد نظر باشد. این کلاس در namespace ای براساس اسم پکیج تعریف شده است. علت این نام گذاری این ا ست که وقتی از کلاسِ پیامی در C++ می خواهیم ا ستفاده کنیم، از (::) ا ستفاده

می کنیم، که اسمِ پکیج را از اسمِ نوعِ داده جدا می کند. در مثال pubvel، سربرگ کلاسی با نام geometry_msgs::Twist را معرفی می کند.

ایجاد شیء منتشر کننده

در واقع کار انتشار پیام به وسیله ی یک شی از کلاس انجام می شود.[1] خطی مانند زیر شی مورد نیاز را ایجاد می کند:

```
ros::Publisher pub = node_handle.advertise<message_type>(topic_name, queue_size);
```

بیاید به هر قسمت این خط نگاهی بیندازیم:

- node_handle یک شی از کلاس ros::NodeHandle است، که می توانید در ابتدای برنامه یتان ایجاد کنید. ما با متد advertise این شی را فراخوانی خواهیم کرد.
- قسمت message_type درون براکت زاویه دار ـــ که قالب پارامتر نام دارد - نوعِ داده ی پیامی است که می خواهیم منتشر کنیم و باید اسم کلاسی که در سربرگ در بالا تعریف کردیم باشد. در این مثال کلاس geometry_msgs::Twist را استفاده می کنیم.
- topic_name رشته ای شامل اسم تاپیکی که می خواهیم منتشر کنیم است. این اسم باید بمانند اسامی تاپیک نشان داده شده در rostopic list یا rqt_graph باشد. اما معمولاً بدون اسلش /. ما اسلش / را در ابتدا حذف می کنیم که یک اسم تاپیک نسبی ایجاد کنیم؛ در فصل ۵ مکانیزم و اهداف اسم های نسبی را بیان می کنیم. در این مثال، اسم تاپیک "turtle1/cmd_vel" است.

مواظب تفاوت بین اسم تاپیک و نوع پیام باشید. اگر شما اشتباهاً این دو را عوض کنید، خطاهای کمپایل گیج کننده ی زیادی را ایجاد می کنید.

- آخرین پارامتر برای advertise یک عدد صحیح، نشان دهنده ی سایز صف نگهدارنده ی پیام (تعداد پیام هایی که منتشر کننده می تواند نگه دارد)، برای این ناشر است. در بیشتر موارد، یک عدد بزرگی مانند ۱۰۰۰ برای این قسمت مناسب است. اگر برنامه ی شما پیام ها را سریعتر از سایز صف نگهدارنده منتشر کند، قدیمی ترین پیام دور انداخته می شود.

این پارامتر در بیشتر مواقع مورد نیاز است چون در بیشتر موارد، پیام باید به نود دیگری فرستاده می شود. این پروسه ارتباط می تواند زمان گیر باشد، بخصوص نسبت به زمانی که برای ایجاد پیام نیاز است. رآس این تأخیر را با داشتن روش انتشار - ذخیره ی پیام در صف خارجی و بلافاصله برگشتن - کاهش می دهد. در واقع یک ترد (thread) جداگانه در پشت صحنه پیام

[1] http://wiki.ros.org/roscpp/Overview/PublishersandSubscribers

را انتقال می دهد. مقدار عدد صحیح تعداد پیام هایی - نه تعداد بایت ها - که صف نگه دارنده ی پیام نگه می دارد را نشان می دهد.
جالب ا ست که کتابخانه های مورد ا ستفاده ی رآس اینقدر هو شمند ه ستند که می دانند چه زمانی نودهای منتشر کننده و شنونده قسمتی از یک پروسه هستند. در این مورد بدون استفاده از هیچ شبکه ی انتقالی پیام ها مستقیماً توسط شنونده دریافت می شوند. این ویژگی برای کارآمد ساختن nodelets[1] بسیار مهم است. نودلت چند نود است که می توانند به صورت داینامیک در یک پروسه ی (بدون کپی کردن پیام ها) استفاده بشوند.

برای منتشر کردن پیام ها از یک نود روی چند تاپیک، با ید برای هر تاپیک یک شی ros::Publisher جدا ایجاد کنید.

مواظب طول عمر ros::Publisher با شید. ایجاد یک نا شر یک عمل سنگین ا ست، بنابراین ایده بدی ا ست که برای منتشر کردن هر پیام یک ros::Publisher جدیدایجاد کنیم. به جایش برای هر تاپیک یک ناشر ایجاد می کنیم و در طول قسمت اجرایی برنامه از آن ناشر استفاده می کنیم. در pubvel، این کار را با تعریف ناشر خارج از حلقه ی while انجام داده ایم.

ایجاد و پرکردن شی پیام: در قدم بعد، خود شی پیام را می سازیم. قبلا هم به کلاسِ پیام اشاره کردیم، وقتی داشتیم شی ros::Publisher را ایجاد می کردیم. شی های این کلاس دارای یک عضو قابل دسترس عموم برای هر قسمت از نوع پیام است.

ما با ا ستفاده از rosmsg (در بخش ۲,۷,۲) دیدیم که نوعِ پیام geometry_msgs/Twist دارای دو لایه ی بالایی (linear و angular) ا ست که هر کدام دارای زیر شاخه ی (x و y و z) ه ستند. و هر کدام از این زیرشاخه ها یک عدد اعشاری ۶۴ بیتی است که در کامپایلرهای C++ یک double خوانده می شود. کد لیست ۳,۴ شی geometry_msgs::Twist را ایجاد می کند و دو عدد تصادفی (pseudo-random) را به دوتا از اعضای آن نسبت می دهد.

```
geometry_msgs::Twist msg;
msg.linear.x = double(rand())/double(RAND_MAX);
msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
```

این کد سرعت خطی را عددی بین صفر و یک انتخاب می کند، و سرعت زاویه ای را عددی بین ۱- و ۱. چون چهار قسمت دیگر را (msg.linear.y ,msg.linear.z , msg.angular.x , and msg.angular.y) نادیده می گیرد، ما هم مقدار پیش فرض که صفر است را تغییر ندادیم.
حتماً بیشتر نوع پیام ها دارای نوع دیگری غیر از float64 هستند. خوشبختانه نگاشت از یک نوع در رآس به نوع C++ همان طور که انتظار دارید به خوبی کار می کند.[2] یک مورد که ممکن ا ست

[1] http://wiki.ros.org/nodelet

[2] http://wiki.ros.org/msg

خیلی واضح نباشد این است که نوع قسمت هایی از آرایه (که در rosmsg show با براکت نشان داده می شود) در کد C++ به عنوان آرایه ی STL شناخته می شود.

انتشار یک پیام :بعد از انجام کارهای اولیه، انتشار یک پیام بسیار ساده است، با استفاده از متد publish از شی ros::Publisher. در این مثال، به صورت زیر:

```
pub.publish(msg);
```

این متد msg داده شده را به صف نگهدارنده ناشر اضافه می کند، که از آن صف در اسرع وقت به شنونده ی مربوطه فرستاده می شود.

فرم دادن خروجی :گرچه این مورد به صورت مستقیم به منتشر کردن دستور سرعت مربوط نیست، نگاهی به خط ROS_INFO_STREAM در لیست ۳,۴ ارزشمند است. اینجا عملکرد ROS_INFO_STREAM روشن تر است، چون توانایی وارد کردن داده هایی غیر از string را در خروجی نشان می دهد - در این مورد، وارد کردن قسمتی از پیام که به صورت تصادفی مشخص شده. بخش ۴,۳ اطلاعات بیشتری از عملکرد ROS_INFO_STREAM به دست می دهد.

۳-۳-۲- انتشار در حلقه

بخش قبل جزیئات انتشار پیام را پوشش داد. در مثال pubvel مرحله انتشار پیام در حلقه ی while تکرار می شود تا با گذشت زمان پیام های متفاوتی منتشر شود. برنامه ازدو ساختار اضافه دیگر برای شکل دادن حلقه استفاده می کند.

چک کردن برای خاموش کردن نود: شرط حلقه ی while در pubvel:

```
ros::ok()
```

به صورت غیر مستقیم این تابع چک می کند که آیا برنامه ی ما در شرایط خوبی به عنوان یک نود رآس هست یا نه. این تابع تا زمانی که نود به هر دلیلی خاموش نشود، true برمی گرداند. چند روش برای اینکه ros::ok() جواب false برگرداند:

- شما می توانید برای نود از rosnode kill استفاده کنید.
- شما می توانید علامت توقف Ctrl-C را برای برنامه بفرستید.

یک موضوع جالب، ros::init() یک handler برای علامت Ctrl-C نصب می کند، و برای ایجاد یک خاموش کردن راحت استفاده می کند. تأثیر Ctrl-C این است که ros::ok() پاسخ false برگرداند، اما برنامه را یک دفعه نابود نمی کند. این مورد احتمالاً مهم است اگر مراحل پاک

سازی مانند نو شتن فایل لوگ، ذخیره کردن قسمتی از نتایج، پیام خداحافظی فر ستادن و ... قبل از اتمام برنامه وجود داشته باشد.

- شما در خود برنامه می توانید دستور زیر را فرا بخوانید:

ros::shutdown()

این تابع احتمالاً روش مفیدی است که از داخل برنامه نشان بدهید کار نود شما تمام شده است.

- شما می توانید نود دیگری را با همین نام شروع کنید. این معمولاً زمانی اتفاق می افتد که شما نمونه ی جدیدی از همان برنامه را اجرا می کنید.

کنترل سرعت انتشار : آخرین المان جدید pubvel استفاده از شی ros::Rate است:[1]

ros::Rate rate(2);

این شـی سـرعت اجرای حلقه را کنترل می کند. واحد پارامتر سـازنده ی (constructor) آن هرتز Hz (تعداد تکرار حلقه در ثانیه) است. در این مثال شی ای ساخته شده است که حلقه را دو بار بر ثانیه اجرا می کند. در آخر هر حلقه، متد sleep را از این شیء فراخوانی می کنیم:

rate.sleep();

فراخوانی این متد باعث ایجاد تأخیر در برنامه می شـود. مدت تأخیر طوری محاسـبه می شـود که اجرای حلقه سریعتر از سرعت تعریف شده نشود. بدون این نوع کنترل، برنامه پیام را با بیشترین سـرعتی که کامپیوتر اجازه می دهد منتشـر می کند، که ممکن اسـت صـف نگهدارنده ی ناشـر و شنونده را ا شباع کند و اتلاف منابع شبکه و محا سباتی ا ست. (در کامپیوتر نویسنده یک ویرایش برنامه حدوداً ۶۳۰۰ پیام در ثانیه منتشر می کند.)
شـما می توانید با اسـتفاده از rostopic hz این تنظیمات را تأیید کنید. نتیجه ی pubvel باید به صورت زیر باشد:

average rate: 2.000
min: 0.500s max: 0.500s std dev: 0.00006s window: 10

می بینید که پیام ما با سـرعت دو بار بر ثانیه (با کمی اختلاف از برنامه ی زمانی اش) منتشـر می شود.

شما ممکن ا ست به روش جایگزینی برای ros::Rate فکر می کنید، ا ستفاده از تأخیری ثابت با ا ستفاده از sleep یا usleep در هر تکرار حلقه. مزیت ا ستفاده از شی ros::Rate نسبت به این روش این است که ros::Rate می تواند زمان مورد استفاده بقیه قسمت های حلقه را نیز در نظر

[1] http://wiki.ros.org/roscpp/Overview/Time

بگیرد. اگر محاسبات مهم محدودی باید در هر تکرار انجام شود (همان طور که از هر برنامه ی واقعی انتظار می رود)، زمان محاسباتی مورد استفاده از تأخیر کم می شود. در حالت حاد که کار واقعی درون حلقه بیشتر از سرعت مورد نظر طول بکشد، تأخیر ()sleep به صفر کاهش می یابد.

۳-۳-۳- کمپایل کردن pubvel

فرآیند کمپایل کردن pubvel تقریباً مشابه hello است: اصلاح CMakeLists.txt و package.xml، استفاده از catkin_make برای کمپایل کردن فضای کاری. با این وجود یک تفاوت مهم بینشان وجود دارد.

م شخص کردن واب ستگی به نوعِ پیام : چون pubvel از نوعِ پیامی از پکیج geometry_msgs ا ستفاده می کند باید این پکیج به عنوان واب ستگی ذکر شود. به مانند واب ستگی به roscpp که در بخش ۳,۲,۲ بحث کردیم. مشخصاً ما باید خط find_package را در CMakeLists.txt اصلاح کنیم و geometry_msgs را بعد از roscpp اضافه کنیم:

```
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs)
```

توجه داشته باشید که بجز اصلاح find_package موجود، ما دو خط جدید به package.xml اضافه می کنیم، که در آن باید المان های زیر را برای وابستگی جدید اضافه کنیم:

```
<build_depend>geometry_msgs</build_depend>
<run_depend>geometry_msgs</run_depend>
```

اگر شما این قدم را فراموش کنید، احتمالاً catkin_make نمی تواند سربرگ geometry_msgs/Twist.h را پیدا کند. وقتی شما خطایی مبنی بر پیدا نکردن سربرگی دیدید، بهتر است وابستگی های پکیج خود را بازرسی کنید.

۳-۳-۴- اجرای pubvel

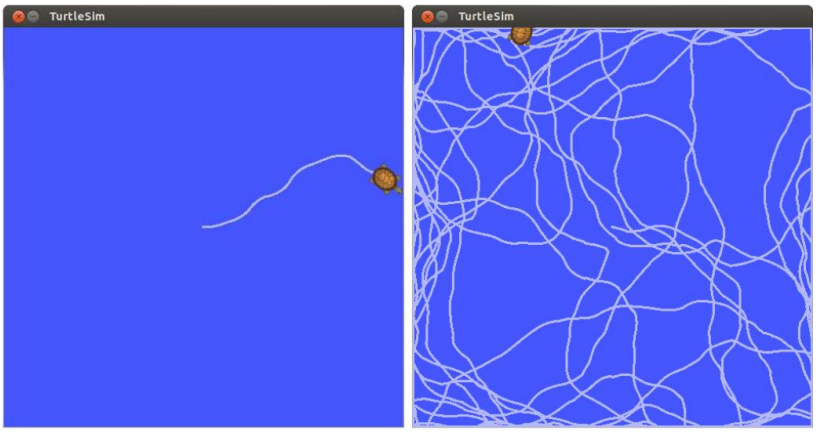
در آخر ما آماده ی اجرای pubvel هستیم. مثل همیشه rosrun می تواند این کار را انجام دهد.

```
rosrun agitr pubvel
```

شاید شما مایل با شید turtlesim را نیز اجرا کنید تا عکس العمل لاک پشت را به د ستور حرکت که pubvel منتشر می کند ببینید:

rosrun turtlesim turtlesim_node

شکل ۳,۱ یک نمونه از نتیجه را نشان می دهد.



شکل (۳-۱) عکس العمل لاک پشت turtlesim به دستور سرعت تصادفی از pubvel

۳-۴- برنامه ی شنونده (ساب اسکرایبر)

تا کنون ما یک مثال از برنامه ی منتشـــرکننده ی پیام دیدیم. مســـلماً وقتی در مورد ارتباط بین نودها صـحبت می کنیم، این نصـف ماجرا اسـت. حالا بیاید برنامه ای را بررسـی کنیم که به پیام منتشر شده از نودهای دیگر گوش می دهد.[۱]

اســـتفاده از turtlesim را ادامه می دهیم، به تاپیک turtle1/pose/ گوش خواهیم داد، که آن را turtlesim_node منتشر می کند. (چگونه بفهمیم turtlesim_node این تاپیک را منتشر می کند؟ یک راه پیدا کردنش شـروع نود و اسـتفاده از یکی از دسـتورات , rosnode info , rostopic list rqt_graph است که ببینیم چه تاپیکی منتشر شده است. بخش ۲,۷,۱) پیام این تاپیک موقعیت، مکان و زاویه ی، لاک پشت را نشان می دهد. لیست ۳,۵ یک برنامه ی کوچک که به این پیام ها گوش می کند را نشان می دهد و آنها را برای ما به وسیله ی ROS_INFO_STREAM خلاصه می کند. گرچه بخش هایی از این برنامه باید تا حالا آشنا باشند، سه اصل جدید وجود دارد.

[۱] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(C++)

```
1  // This program subscribes to turtle1/pose and shows its
2  // messages on the screen.
3  #include <ros/ros.h>
4  #include <turtlesim/Pose.h>
5  #include <iomanip> // for std::setprecision and std::fixed
6
7  // A callback function.  Executed each time a new pose
8  // message arrives.
9  void poseMessageReceived(const turtlesim::Pose& msg) {
10   ROS_INFO_STREAM(std::setprecision(2) << std::fixed
11     << "position=(" <<  msg.x << "," << msg.y << ")"
12     << " direction=" << msg.theta);
13 }
14
15 int main(int argc, char **argv) {
16   // Initialize the ROS system and become a node.
17   ros::init(argc, argv, "subscribe_to_pose");
18   ros::NodeHandle nh;
19
20   // Create a subscriber object.
21   ros::Subscriber sub = nh.subscribe("turtle1/pose", 1000,
22     &poseMessageReceived);
23
24   // Let ROS take over.
25   ros::spin();
26 }
```

لیست (۳-۵) برنامه ای از رآس به نام subpose.cpp که به داده موقعیت منتشر شده از ربات turtlesim گوش می دهد.

نوشتن یک تابع callback : یک تفاوت مهم ناشر و شنونده این است که نود شنونده نمی داند چه زمانی پیام را دریافت خواهد کرد. بر این اساس، ما باید همه ی کدهایی را که پاسخی به پیام دریافت شده هستند درون تابعی به نام callback قرار دهیم، که این تابع با هر پیامی که دریافت می شود فراخوانده می شود. یک تابع callback شنونده به صورت زیر است:

```
void function_name(const package_name::type_name &msg) {
...
}
```

package_name و type_name مشابه ناشر هستند: آنها به کلاس پیام تاپیکی که ما تصمیم داریم به آن گوش کنیم ارجاع می دهند. بدنه ی تابع callback به همه ی قسمت های پیام دریافت شده دسترسی دارند و می تواند آنها را ذخیره، استفاده یا نادیده بگیرد. مانند همیشه ما باید سربرگ هایی را که این کلاس را تعریف می کنند اضافه کنیم.

در این مثال، callback پیامی با نوع turtlesim::Pose را در یافت می کند، بنابراین ما به turtlesim/Pose.h نیاز داریم. (با استفاده از rostopic info می توانیم چک کنیم که این نوع پیام درست است؛ بخش ۲,۷,۲ را دوباره نگاه کنید.) callback با استفاده از ROS_INFO_STREAM تعدادی از پیام ها، شامل عضوهای y،x و theta را پرینت می کند. (ما می توانیم با استفاده از rosmsg show قسمت های نوعِ پیام را چک کنیم، با استفاده از بخش ۲,۷,۲.) یک برنامه ی واقعی، قطعاً کار بامعنایی با این پیام ها می کند.

توجه داشته باشید، تابع های callback شنونده چیزی را برنمی گردانند و از نوع void return هستند. با کمی تأمل این مطلب برای ما منطقی خواهد بود، چون وظیفه ی رآس است که این تابع را فرابخواند، جایی در برنامه ی ما برای استفاده از هیچ مقداری وجود ندارد.

ایجاد یک شی شنونده: برای شنیدن یک تاپیک، ما یک ros::Subscriber ایجاد می کنیم:[1]

```
ros::Subscriber sub = node_handle.subscribe(topic_name,queue_size, pointer_to_callback_function);
```

این خط قسمت های جابه جا پذیری دارد (بیشتر قسمت ها به مانند دستور ros::Publisher هستند):

- node_handle به مانند نود نگهدارنده ای است که تا کنون دیده ایم.
- topic_name نام تاپیکی است که به آن می خواهیم گوش کنیم، به صورت یک متغییر رشته ای (string). در این مثال "turtle1/pose" استفاده شده است. دوباره در اینجا ما اسلش / را حذف کرده ایم که یک نام نسبی داشته باشیم.
- queue_size سایز (به صورت عدد صحیح) صف پیام برای گوش دادن است. معمولاً شما می توانید عدد بزرگی مانند ۱۰۰۰ را بدون نگرانی برای پروسه ی صف استفاده کنید.

وقتی یک پیام ها دریافت می شوند، در یک صف ذخیره می شوند تا زمانی که رآس شانس اجرای تابع callback را داشته باشد. این پارامتر حداکثر تعداد پیامی که رآس می تواند در صف نگه دارد را مشخص می کند. اگر یک پیام دریافت شود در حالیکه صف پر است، قدیمی ترین پیامی که پردازش نشده حذف می شود تا فضای خالی ایجاد شود. این امر در ظاهر به مانند تکنیکی که برای انتشار پیام استفاده می شد می باشد -صفحه ی ۴۶ را ببینید- اما تفاوت مهمی دارد: سرعت رآس برای خالی کردن صف ناشر وابسته به زمانی است که پیام از ناشر به شنونده منتقل می شود و تقریباً خارج از کنترل ماست. برعکس، سرعت خالی کردن صف شنونده وابسته به این است که ما چقدر سریع پردازش می کنیم. بنابراین، ما می توانیم احتمال لبریز شدن صف شنونده را کاهش دهیم با الف) با اطمینان از اینکه callback به طور مدام

[1] http://wiki.ros.org/roscpp/Overview/PublishersandSubscribers

انجام می شود با ا ستفاده از ros::spin و ros::spinOnce ب) با کاهش زمان مورد ا ستفاده هر callback.

- آخرین پارامتر یک ا شاره گر به تابع callback ا ست که رآس باید آن را وقتی اجرا کند که پیامی ر سید. در C++، شما می توانید ا شاره گر به یک تابع را با ا ستفاده از علامت اپراتور (& , "آدرسش") قبل از نامش بسازید. در مثال ما به شکل زیر:

&poseMessageReceived

ا شتباه معمول نو شتن () یا حتی (پیام) را بعد از ا سم تابع نکنید. این پرانتزها (عبارت ها) را در واقع ز مانی نیاز دار ید که می خواهید تابع را فراخوانی کنید، نه وقتی می خواهید بدون فراخواندن تابع به آن ا شاره کنید(همان کاری که ما اینجا انجام می دهیم). رآس عبارتهای مورد نیاز را زمان فراخواندن تابع فراهم می کند.

پیشنهادی در مورد متن C++ : علامت ها در واقع دلخواه هستند، و بیشتر برنامه ها آن را حذف کرده اند. کامپایلر می تواند بگوید که شما می خواهید به یک تابعی ا شاره کنید، به جای اینکه مقداری را از اجرای تابع بازگرداند، چون بعد از اسم تابع پرانتز نیآمده است. اما نویسنده پیشنهاد می کند علامت & را اضافه کنید، چون در واقع برای خواننده واضح تر خواهد بود که ما اینجا اشاره گر داریم.

شاید شما توجه کرده باشید که وقتی یک شیء ros::Subscriber می سازیم، نوعِ پیام را به صورت وا ضح بیان نمی کنیم. در واقع متد advertise یک الگو ا ست که کمپایلر C++ نوعِ پیام در ست را براساس نوع داده تابع callback که به آن اشاره کردیم تعیین می کند.

اگر شما نوعِ پیام ا شتباهی را در تابع callback ا ستفاده کرده با شید، کامپایلراین خطا را پیدا نخواهد کرد. به جای آن شما خطای run-time را در مورد عدم تطابق نوع دریافت خواهید کرد. این خطا ممکن است وابسته به تنظیم زمان و برآمده از نود ناشر یا شنونده باشد.

یک نکته که می تواند در مورد شی های ros::Subscriber غیرشهودی باشد این است که تقریباً به ندرت هیچ یک از این متدهایشان را فرا می خوانند. در عوض زمان عمر یک شی مهمترین قسمت مربوطه ا ست: وقتی ما یک ros::Subscriber را می سازیم، نود ما ارتباطی با همه ی نا شرها به نام آن تاپیک را ایجاد می کند. وقتی شئ از بین می رود، چه با متوقف شدن، چه با حذف شدن، با ایجاد یک شئ توسط یک اپراتور جدید، ارتباطات هم از بین می روند.

کنترل رآس : آخرین مسأله این است که رآس فقط زمانی تابع callback ما را اجرا می کند که به آن مشخصاً اجازه داده باشیم.[1] در واقع دو راه کمی متفاوت برای انجام این کار وجود دارد. یک روش بدین صورت است:

```
ros::spinOnce();
```

این دستور از رآس می خواهد که همه ی callback های منتظرِ نودهای شنونده را اجرا کند و بعد دوباره کنترل را به ما برگرداند. روش دیگر بدین شکل است:

```
ros::spin();
```

این خط به جای ros::spinOnce() از رآس می خواهد که اینجا منتظر بماند و تا زمان خاموش شدن نود callback را اجرا کند. به عبارت دیگر، ros::spin() تقریباً شبیه حلقه ی زیر است:

```
while(ros::ok()) {
ros::spinOnce();
}
```

سؤالی که مطرح می شود این است که از ros::spinOnce() یا از ros::spin() استفاده کنیم: آیا برنامه ی شما یک کار دیگر به جزء پاسخ به callback ها را به صورت متناوب انجام میدهد؟ اگر جواب نه است پس از ros::spin استفاده کنید. اگر بله، پس منطقی است که یک حلقه برای انجام کارهای دیگر و شامل فراخواندن ros::spinOnce() برای انجام callback ها به صورت متناوب بنویسید. لیست ۳,۵ از ros::spin() استفاده کرده است چون تنها وظیفه ی این برنامه دریافت و خلاصه کردن پیام موقعیت دریافت شده است.

یک خطای متداول در برنامه های شنونده این است که هر دو دستور ros::spinOnce و ros::spin اشتباهاً حذف شده باشند. در این مورد، رآس هیچ وقت تابع callback شما را اجرا نخواهد کرد. حذف ros::spin باعث خواهد شد، مدتی کوتاه بعد از شروع برنامه، از برنامه خارج شوید. حذف ros::spinOnce باعث خواهد شد، شما پیامی را دریافت نکنید.

```
[INFO ] [1370972120.089584153]: position =(2.42 ,2.32) direction =1.93
[INFO ] [1370972120.105376510]: position =(2.41 ,2.33) direction =1.95
[INFO ] [1370972120.121365352]: position =(2.41 ,2.34) direction =1.96
[INFO ] [1370972120.137468325]: position =(2.40 ,2.36) direction =1.98
[INFO ] [1370972120.153486499]: position =(2.40 ,2.37) direction =2.00
[INFO ] [1370972120.169468546]: position =(2.39 ,2.38) direction =2.01
[INFO ] [1370972120.185472204]: position =(2.39 ,2.39) direction =2.03
```

لیست (۳-۶) قسمتی از خروجی subpose، که تغییر تدریجی موقعیت ربات را نشان می دهد.

[1] http://wiki.ros.org/roscpp/Overview/CallbacksandSpinning

۳-۴-۱- کامپایل کردن و اجرای برنامه ی subpose

این برنامه می تواند مانند دو مثال قبل کامپایل و اجرا شود.

مطمئن شوید که پکیج شما شامل وابستگی به turtlesim است، ما به این پکیج نیاز دارد چون از نوعِ پیام turtlesim/Pose ا ستفاده می کنیم. بخش ۳,۳,۳ را می توانید جهت یادآوری چگونه بیان وابستگی ها دوباره ببنید.

قسمتی از خروجی برنامه، وقتی دو نود turtlesim_node و pubvel اجرا شده اند، را در لیست ۳,۶ می بینید.

۳-۵- در ادامه

هدف از این فصل نوشتن، کمپایل کردن، و اجرای چند برنامه بود، شامل برنامه هایی که عملیات های اصلی رآس (منتشر کردن و گوش کردن) را اجرا می کنند. هر کدام از این برنامه ها با استفاده از ماکرو به نام ROS_INFO_STREAM پیام های اطلاعاتی برای لاگ ایجاد می کنند. در بخش بعدی، سیستم ورودی رآس را بررسی می کنیم، که ROS_INFO_STREAM تنها قسمت کوچکی از آن بود.

فـصل ۴ : پیام های لاگ

در این فصل ما پیامهای لاگ را ایجاد می کنیم و می بینیم.

تا اینجا دیدیم (در مثال برنامه ی فصل ۳) یک ماکرو به نام ROS_INFO_STREAM پیام های اطلاعاتی را به کاربر نشان می دهد. این پیام قسمتی از پیام های لاگ است. رآس سیستم ثبت وقایع (logging) قدرتمندی دارد که شامل ROS_INFO_STREAM و تعداد دیگری از ویژگی هاست. در این فصل چگونگی استفاده از این سیستم لاگینگ را یاد می گیریم.

۴-۱- سطح شدت

ایده ی سیستم لاگینگ رآس ـــ به طور کلی نرم افزار لاگینگ در بیشتر جاها ـــ به برنامه ها این امکان را می دهد که یک رشته ی متنی کوتاه به نام پیام لاگ را ایجاد کند. در رآس، پیامهای لاگ به پنج گروه به نام سطح شدت دسته بندی می شوند، که گاهی تنها شدت ها و گاهی فقط سطح ها خوانده می شوند. سطح ها به ترتیب اهمیتشان عبارت اند از:[۱]

- DEBUG (اشکال زدایی)
- INFO (اطلاعات)
- WARN (اخطار)
- ERROR (خطا)
- FATAL(مهلک)

پیام DEBUG (اشکال زدایی) ممکن است مکرراً ایجاد شود، اما به طور کلی در زمانی که برنامه به درستی کار می کند علاقه ای به آنها نداریم. پیام های FATAL (مهلک) در انتهای لیست ممکن است به ندرت ایجاد شوند اما بسیار مهم هستند و مشکلی را نشان می دهد که برنامه را از ادامه بازمی دارد. سه سطح دیگر INFO و WARN و ERROR (اطلاعات و اخطار و خطا) در سطحی بین این دو درجه از اهمیت هستند. در ادامه مثالی از رآس برای هر یک از این سطح شدت ها نشان داده شده است.

[۱] http://wiki.ros.org/VerbosityLevels

مثال	شدت
خواندن سربرگ از بافر	DEBUG (اشکال زدایی)
انتظار برای برقرار شدن همه ی ارتباطات	INFO (اطلاعات)
کمتر از ۵ گیگا بایت GB حافظه	WARN (اخطار)
سربرگ ناشر نیازی به المان type ندارد.	ERROR (خطا)
شما باید ros::init() قبل از ساختن اولین NodeHandle فرا بخوانید.	FATAL (مهلک)

این تنوع سطح شدت قصد دارد روش ثابتی برای دسته بندی و مدیریت پیام های لاگ را فراهم کند. در ادامه چگونه فیلتر کردن و برجسته کردن پیام ها براساس سطح شدتشان را در یک مثال خواهیم دید. گرچه، سطح ها خود شان معنایی را به دنبال ندارند: ایجاد شدن یک پیام FATAL (مهلک) باعث پایان یافتن برنامه نمی شـود. همین طور، ایجاد کردن یک پیام DEBUG (اشـکال زدایی) ، اشکال برنامه ی شما را رفع نخواهد کرد.

۴-۲- یک برنامه

در ادامه ی این فصل خواهیم دید که چگونه پیامهای لاگ را بسازیم و ببینیم. مانند همیشه، یک برنامه ی آماده برای روشن کردن موضوع بسیار مفید است. می توانیم از برنامه ی turtlesim در شرایط مناسب برای این هدف استفاده کنیم. turtlesim_node پیامهای لاگ در همه ی سطح ها به جزء پیام FATAL (مهلک) ایجاد خواهد کرد اما برای یادگیری خیلی راحتتر است که با برنامه ای کار کنیم که پیام های لاگ بسیاری در زمان های مشخصی را ایجاد می کند.
لیست ۴,۱ برنامه ای براساس این توصیفات را نشان می دهد. پیامهایی در همه ی سطح ها ایجاد می کند. مثالی از خروجی کنسول در لیست ۴,۲ آورده شده است. ما در ادامه ی فصل برنامه را اجرا شده فرض می کنیم.

```
1   // This program periodically generates log messages at
2   // various severity levels.
3   #include <ros/ros.h>
4   int main(int argc, char **argv) {
5     // Initialize the ROS system and become a node.
6     ros::init(argc, argv, "count_and_log");
7     ros::NodeHandle nh;
8     // Generate log messages of varying severity regularly.
9     ros::Rate rate(10);
10    for(int i = 1;ros::ok();i++) {
11      ROS_DEBUG_STREAM("Counted to " << i);
12      if((i % 3) == 0) {
13        ROS_INFO_STREAM(i << " is divisible by 3.");
14      }
15      if((i % 5) == 0 ) {
16        ROS_WARN_STREAM(i << " is divisible by 5.");
17      }
18      if((i % 10) == 0) {
19        ROS_ERROR_STREAM(i << " is divisible by 10.");
20      }
21      if((i % 20) == 0) {
22        ROS_FATAL_STREAM(i << " is divisible by 20.");
23      }
24      rate.sleep();
25    }
26  }
```

لیست (۴-۱) برنامه ی count.cpp که پیامهای لاگی در هر پنج سطح ایجاد می کند.

```
1  [ INFO ]   [1375889196.165921375]: 3 is divisible by 3.
2  [ WARN]    [1375889196.365852904]: 5 is divisible by 5.
3  [ INFO ]   [1375889196.465844839]: 6 is divisible by 3.
4  [ INFO ]   [1375889196.765849224]: 9 is divisible by 3.
5  [ WARN]    [1375889196.865985094]: 10 is divisible by 5.
6  [ERROR]    [1375889196.866608041]: 10 is divisible by 10.
7  [ INFO ]   [1375889197.065870949]: 12 is divisible by 3.
8  [ INFO ]   [1375889197.365847834]: 15 is divisible by 3.
```

لیست (۴-۲) قسمتی از خروجی برنامه ی count برای چند ثانیه. این برنامه پیام سطح DEBUG را ندارد، چون به صورت پیش فرض پایین ترین سطح INFO است.

۴-۳- ایجاد پیام های لاگ

بیاید نگاه کاملتری به چگونگی ایجاد پیام های لاگ در C++ داشته باشیم.

ایجاد یک پیام لاگ ساده : پنج ماکرو پایه ای در C++ برای ایجاد پیام های لاگ برای هر سطح شدت وجود دارد:

```
ROS_DEBUG_STREAM(message);
ROS_INFO_STREAM(message);
ROS_WARN_STREAM(message);
ROS_ERROR_STREAM(message);
ROS_FATAL_STREAM(message);
```

آرگومانِ پیام هر کدام از این ماکرو ها می توانند تمام عباراتی که ostream در C++ دارد را پیشتیبانی کند، مانند std::cout. شامل عملگر (<<) برای داده های از نوع int یا double، نوع های ترکیبی که عملگر به صورت مناسب بارگزاری شده باشد، و برای رشته های کنترلی استانداردی مانند std::fixed و std::setprecision یا std::boolalpha.

رشته های کنترلی فقط روی پیام های لاگ که نشان داده می شود تأثیر می گذارد. بنابراین هر بار که مایل باشیم باید دوباره رشته های کنترلی را استفاده کنیم.

چرا این محدودیت برای رشته های کنترلی وجود دارد: همان طور حروف بزرگ نشان می دهند، ساختار ROS_. . . _STREAM ماکرو است. هر کدام کد کوتاهی است که std::stringstream را ایجاد می کند و آرگومان هایی که شما فراهم کرده اید را درون رشته ی مورد نظر وارد می کند. بعد این کد توسعه یافته این متن شکل گرفته ی std::stringstream را به سیستم لاگینگ log4cxx منتقل می کند.[۱] وقتی پرو سه تمام شود std::stringstream از بین می رود و حالت داخلی اش، شامل تمام تنظیمات برپا شده ی آن به وسیله ی رشته ی کنترلی، از بین می رود.

اگر شما شیوه ی printf را به جای شیوه ی C++ ترجیح می دهید، ماکروی که پسوند _STREAM از آن حذف شده است وجود دارد، برای مثال، ماکرو

```
ROS_INFO(format, . . . );
```

پیام لاگ در سطح INFO ایجاد می کند. اگر شما با printf آشنا باشید این ماکروها دقیقاً آنطور که شما انتظار دارید کار می کنند. به عنوان مثال، خروجی لیست ۳,۴ تقریباً برابر است با:

```
ROS_INFO("position=(%0.2f,%0.2f)  direction=%0.2f",msg.x,  msg.y,
                                                    msg.theta);
```

[۱] http://wiki.apache.org/logging-log4cxx/

همچنین ماکروهای به شـــیوه ی printf یک بار (ONCE_ . . .) و توقف (throttled_ . . .) با حذف قسمت STREAM_ نیز در ادامه معرفی شده اند.

توجه داشــته باشــید که نیازی به اســتفاده از std::endl یا ایجاد کننده ی خط جدید وجود ندارد، چون سیستم لاگینگ خودش خط ایجاد می کند. فراخوانی هر کدام از این ماکروها یک پیام لاگ تنها و کامل که در یک خط جداگانه نمایش داده می شود را ایجاد می کند.

ایجاد پیام لاگ برای یک بار one-time: گاهی پیام های لاگی داخل حلقه یا داخل تابعی که مکرراً فراخوانده می شــوند برای کاربر مهم هســتند، اما تکرار ایجاد می کنند. یک راه معمول برای مقابله با این شرایط استفاده از متغییر static است که مطمئن شویم پیام فقط یکبار ایجاد شده است، همان دفعه ی اول که بهش می رسیم. لیست ۴,۳ قسمتی از C++ را که این کار را انجام می دهد نشــان می دهد. برای جلوگیری از تکرار این کد ســنگین، قرار دادنش در یک تابع فایده ندارد، چون در این تکنیک نیاز به متغیر static جداگانه ای برای هر تابع داریم، رآس ماکروهای دســـته کوتاه فراهم کرده است که دقیقاً این نوع پیام های لاگ برای یک بار را ایجاد می کند.

```
ROS_DEBUG_STREAM_ONCE(message);
ROS_INFO_STREAM_ONCE(message);
ROS_WARN_STREAM_ONCE(message);
ROS_ERROR_STREAM_ONCE(message);
ROS_FATAL_STREAM_ONCE(message);
```

دفعه ی اولی که این ماکروها در حین اجرای برنامه در نظر گرفته می شوند، آنها پیام لاگی مشابه به ویرایش های بدون ONCE را ایجاد می کنند. بعد از دفعه ی اول، این عبارت تأثیری ندارد. لیســت ۴,۴ یک مثال کوچک را نشــان می دهد که ماکروهای لاگینگ در اولین بار حلقه هر کدام یک پیام را ایجاد می کنند، بعد از آن نادیده گرفته می شوند.

```
1  // Don't do this directly. Use ROS_. . ._STREAM_ONCE instead .
2  {
3  static bool first_time = true ;
4  if ( first_time ) {
5  ROS_INFO_STREAM( " Here's  some important information"
6  << " that  will only appear once.");
7  first_time = false;
8  }
9  }
```

لیست (۴-۳) قسمتی از کد C++ که پیام لاگ را بعد از دفعه اول غیرفعال می کند. ماکروROS_. . . _STREAM_ONCE کد مشابه ای را ایجاد می کند.

```
1  // This program generates a single log message at each
2  // severity level.
3  #include <ros/ros.h>
4
5  int main(int argc, char **argv) {
6    ros::init(argc, argv, "log_once");
7    ros::NodeHandle nh;
8
9    while(ros::ok()) {
10     ROS_DEBUG_STREAM_ONCE("This appears only once.");
11     ROS_INFO_STREAM_ONCE("This appears only once.");
12     ROS_WARN_STREAM_ONCE("This appears only once.");
13     ROS_ERROR_STREAM_ONCE("This appears only once.");
14     ROS_FATAL_STREAM_ONCE("This appears only once.");
15   }
16 }
```

لیست (۴-۴) برنامه ی once.cpp که فقط پنج پیام لاگ ایجاد می کند.

ایجاد کردن پیام لاگ متوقف شده (throttled log message): همین طور ماکروهایی برای متوقف کردن سرعت ظاهر شدن پیام در لاگ وجود دارد.

```
ROS_DEBUG_STREAM_THROTTLE(interval, message);
ROS_INFO_STREAM_THROTTLE(interval, message);
ROS_WARN_STREAM_THROTTLE(interval, message);
ROS_ERROR_STREAM_THROTTLE(interval, message);
ROS_FATAL_STREAM_THROTTLE(interval, message);
```

پارامتر interval یک double اســـت که حداقل زمان (به ثانیه) بین دو پیام لاگ را مشـــخص می کند.

هر نمونه از ماکروهای ROS_. . . _STREAM_THROTTLE پیام لاگی خود را در بار اول که اجرا شود ایجاد می کند. اجرای بعدی تا زمان مشخصی غیرفعال می شود. زمان خروج (timeout) به صورت جداگانه برای هر نمونه از ماکروها اندازه گیری می شود (با استفاده از یک متغییر محلی static که زمان آخرین دفعه ی اجرا را ذخیره می کند.).

لیســـت ۴,۵ برنامه ای را نشـــان می دهد که ماکروها را به مانند لیســـت ۴,۱ اســـتفاده می کند. مهمترین تفاوت دو برنامه، مربوط به پیام ها، این اســت که برنامه ی لیســت ۴,۵ زمان محاســباتی بیشتری را صرف می کند، چون به جای توقف در زمان مشخص (sleep) از زمان نمونه برداری می کند تا متوجه شـــود چه زمانی باید پیام جدید را ایجاد کند. این نوع نمونه برداری در برنامه های واقعی معمولاً ایده ی بدی است.

```
1  // This program generates log messages at varying severity
2  // levels, throttled to various maximum speeds.
3  #include <ros/ros.h>
4
5  int main(int argc, char **argv) {
6    ros::init(argc, argv, "log_throttled");
7    ros::NodeHandle nh;
8    while(ros::ok()) {
9      ROS_DEBUG_STREAM_THROTTLE(0.1,
10       "This appears every 0.1 seconds.");
11     ROS_INFO_STREAM_THROTTLE(0.3,
12       "This appears every 0.3 seconds.");
13     ROS_WARN_STREAM_THROTTLE(0.5,
14       "This appears every 0.5 seconds.");
15     ROS_ERROR_STREAM_THROTTLE(1.0,
16       "This appears every 1.0 seconds.");
17     ROS_FATAL_STREAM_THROTTLE(2.0,
18       "This appears every 2.0 seconds.");
19   }
20 }
```

لیست (۴-۵) برنامه C++ به نام throttle.cpp که پیام لاگ متوقف شده (throttled log messages) را نشان می دهد.

۴-۴- دیدن پیام های لاگ

تا اینجا دیدیم چگونه یک پیام لاگ را ایجاد کنیم، اما نگفتیم این پیام ها کجا می روند. در واقع سه مقصد برای پیام های لاگ وجود دارد: هر پیام می تواند در خروجی کنسول ظاهر شود یا به عنوان پیامی از تاپیک rosout باشد یا به عنوان ورودی در فایل لاگ باشد. حالا ببینیم هر کدام را چگونه استفاده کنیم.

۴-۴-۱- کنسول

در ابتدا و به صورت خیلی واضح، پیام های لاگ به کنسول فرستاده می شوند. بخصوص، پیام های DEBUG و INFO در خروجی استاندارد پرینت می شوند، پیام های ERROR،WARN و FATAL به خطاهای استاندارد فرستاده می شوند.[۱]

تفاوت قائل شدن بین خروجی استاندارد و خطای استاندارد اینجا بی معناست، مگر اینکه شما بخواهید یکی از این دو رشته یا هر دو را به فایلی یا پایپی نسبت دهید، در این مورد یک سری پیچیدگی به وجود می آید. نحوه ی متداول نسبت دادن فایل:

```
command > file
```

خروجی استاندارد را به فایل نسبت می دهد اما خطای استاندارد را خیر. برای نسبت دادن همه پیام های لاگ به فایل، باید از دستوری مانند زیر استفاده کنید:

```
command &> file
```

مواظب باشید به هر حال، به خاطر اینکه این دو رشته به صورتهای مختلف بافر می شوند ممکن است پیام نامتعارف ظاهر شود ـــ پیام های DEBUG و INFO دیرتر از زمانی که انتظار می رود ظاهر شوند. شما می توانید با استفاده از دستور stdbuf پیام ها را مجبور کنید به صورت متعارف ظاهر شوند چون خروجی استاندارد را مجبور به استفاده از بافرینگ خطی می کند:

```
stdbuf -oL command &> file
```

در آخر توجه داشته باشید رآس از کد رنگی ANSI در خروجی استفاده می کند، که برای انسانها و نرم افزارهایی که آن را نمی فهمند ظاهری مانند ^[[۰m دارد، حتی اگر خروجی به ترمینالی هم نسبت داده نشده باشد. برای دیدن فایلی شامل این نوع کدها، از دستور زیر استفاده کنید:

```
less -r file
```

فرمت پیام های کنسول: شما می توانید فرمت استفاده شده برای پرینت پیام های لاگ را در کنسول با استفاده از تنظیمات متغیر محیطی ROSCONSOLE_FORMAT تغییر دهید. این متغیرها معمولاً شامل یک یا چند قسمت اسم هستند، هر کدام با یک علامت دلار و آکولاد مشخص شده اند، نشان می دهند که هرکدام از داده های پیام لاگ کجا باید نشان داده شوند. فرمت پیش فرض به صورت زیر است:

```
[${severity}] [${time}]: ${message}
```

[۱] http://wiki.ros.org/roscpp/Overview/Logging

این فرمت ممکن است برای بیشتر کاربردها مفید باشد، اما قسمت های دیگری هم وجود دارد که ممکن است مفید باشند: [1]

- برای ذکر کردن جزئیات در مورد مکان سورس کدهایی که پیام را ایجاد کرده اند، ترکیبی از موارد زیر را استفاده کنید:

${file} , ${line} , ${function}

- برای ذکر کردن نام نودی که پیام لاگ را ایجاد کرده از این قسمت استفاده کنید:

${node}

ابزار roslaunch (فصل ۶) به صورت پیش فرض استاندارد خروجی و استاندارد خطا را از نود ایجاد نمی کند. برای دیدن خروجی از نود لانچ شده، باید حتماً نشانه ی "output="screen را استفاده کنید و یا همه ی نودها را مجبور کنید از آن نشانه در roslaunch با استفاده از پارامتر screen-- در کامند لاین استفاده کنند. (صفحه ی ۸۵)

۴-۴-۲- پیام ها در rosout

علاوه بر اینکه پیام های لاگ در کنسول ظاهر می شوند در تاپیکی به نام rosout/ نیز منتشر می شوند. نوعِ پیام این تاپیک rosgraph_msgs/Log است. لیست ۴,۶ بخش هایی از این نوع داده ، شامل سطح شدت و متا-داده های مربوطه را نشان می دهد.

```
1   byte DEBUG=1
2   byte INFO=2
3   byte WARN=4
4   byte ERROR=8
5   byte FATAL=16
6   std_msgs / Header header
7   uint32 seq
8   time stamp
9   string frame_id
10  byte level
11  string name
12  string msg
13  string file
14  string function
15  uint32 line
16  string [ ] topics
```

لیست (۴-۶) بخش های از نوعِ پیام rosgraph_msgs/Log

[1] http://wiki.ros.org/rosconsole

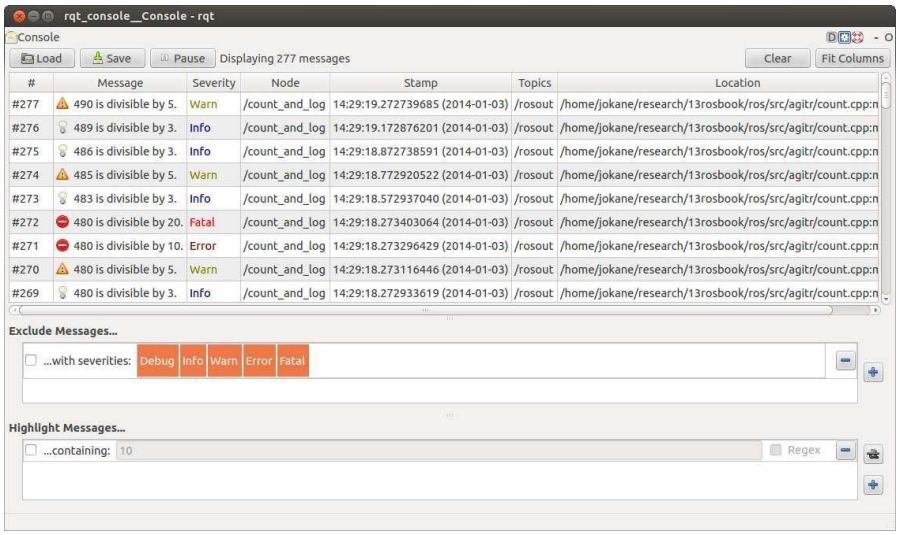
شما ممکن است توجه کرده باشید که اطلاعات هر کدام از این پیام ها کاملاً مشابه خروجی کنسول که در بالا بدان اشاره شد است. اساسی ترین فایده ی rosout/ نسبت به خروجی کنسول، که در یک رشته واحد شامل پیام های لاگ از همه نود های سیستم است. همه ی پیام های لاگ در rosout/ نشان داده می شوند، بدون در نظر گرفتن اینکه نود از کجا، از کی و چگونه شروع شده است و حتی از کدام کامپیوتر اجرا شده است. چون rosout/ یک تاپیک معمولی است مسلماً شما می توانید از دستور زیر برای دیدن پیام به صورت مستقیم استفاده کنید:

```
rostopic echo /rosout
```

اگر مایل باشید می توانید حتی به پیام rosout/ گوش کنید و آن را نمایش یا پردازش کنید. گرچه ساده ترین روش برای استفاده از دیدن rosout/، دستور زیر است:[۱۲]

```
rqt_console
```

شکل ۴,۱ نتیجه ی GUI را نشان می دهد. پیام های لاگ همه ی نودها را در یک خط نشان میدهد، به همراه امکان انتخاب برای پنهان کردن یا برجسته کردن پیام براساس فیلترهای مختلف. در مورد GUI توضیحات بیشتری نیاز نیست.

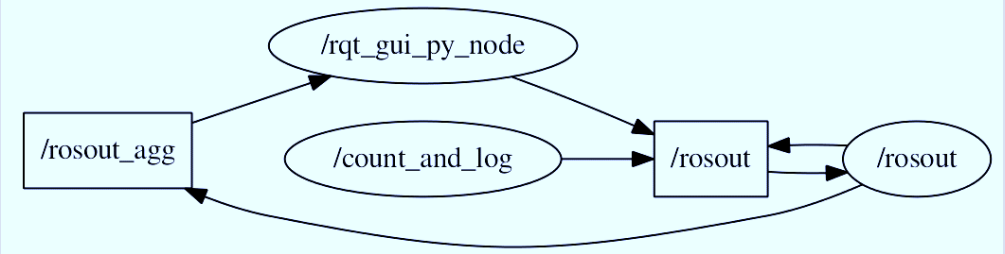


شکل (۴-۱) GUI (رابط گرافیکی) برای rqt_console

توصیف rqt_console در بالا کاملاً درست نیست. در واقع، rqt_console به پیام rosout_agg/ به جای پیام rosout/ گوش می دهد. گراف درست بدین صورت است که وقتی هر دو مثال count و نمونه ی rqt_console/ اجرا شده اند:

[۱] http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch

[۲] http://wiki.ros.org/rqt_console



پسوند agg_ به این واقعیت اشاره می کند که پیام ها از نود rosout جمع شده اند(aggregated). همه ی پیام هایی که در تاپیک rosout/ منتشر شده اند به وسیله نود rosout روی تاپیک rosout_agg/ بازتاب می شوند.

دلیل ظاهری این تکرار، کاهش هزینه دیباگ (اشکال زدایی) است. چون هر نسبت بین ناشر-شنونده باعث ارتباط مستقیم با شبکه ارتباطی بین دو نود می شود، گوش دادن به rosout/ (برای هر نودی که ناشر است) می تواند برای سیستم با نودهای زیاد هزینه بردار باشد، بخصوص وقتی نودها پیام های لاگ زیادی ایجاد کنند. ایده این است که نود rosout تنها نودی است که به rosout/ گوش می دهد و تنها نودی است که آن را روی rosout_agg/ منتشر می کند. ابزارهای اشکال زدایی می توانند به رشته ی پیام های لاگ به طور کامل دسترسی داشته باشند، بدون اینکه با گوش دادن به rosout_agg/ کار اضافه ای برای هر نود در سیستم ایجاد کنند.

از طرف دیگر، پکیج های رآس برای ربات هایی شامل ۲PR و TurtleBot، از الگوی مشابه ای برای پیام های diagnostic استفاده می کنند، که آنها را روی تاپیکی به نام diagnostics/ منتشر می کنند و با نود aggregator روی تاپیک دیگری به نام diagnostics_agg/ بازتاب می دهند.

۴-۴-۳- فایل های لاگ

سومین و آخرین مقصد برای پیام های لاگ، فایل های لاگ ساخته شده توسط نود rosout است. به عنوان قسمتی از تابع callback برای تاپیک rosout/ این نود یک خط درون فایل با نام زیر می نویسد:

~ /.ros/log/run_id/rosout.log

این فایل لاگ rosout.log یک فایل متنی بزرگ است. می توان آن را با ابزار های کامند لاینی مانند head،less یا tail و یا با هر ویرایشگر متنی دلخواه ببینید. run_id یک مشخصه ی جهانی منحصر به فرد (universally-unique identifie -UUID) است که به وسیله آدرس سخت افزاری

MAC کامپیوترتان و زمان فعلی وقتی مستر شروع می شود، ایجاد می شود. یک مثال از run_id بدین صورت است:

57aa1860-d765-11e2-a830-f0def1e189cc

این نوع مشخصه ی منحصر به فرد امکان تشخیص لاگ ها را از دیگر فایل های رآس میّسر می کند.

پیدا کردن run_id : حداقل دو راه ساده برای پیدا run_id وجود دارد.

- شما می توانید خروجی roscore را بررسی کنید. در نزدیک انتهای این خروجی، خطی مانند زیر را می بینید.

setting /run_id to run_id

- شما می توانید از مستر run_id فعلی را بپرسید با دستور زیر:

rosparam get /run_id

این دستور کار می کند چون run_id در پارامترهای سرور ذخیره شده است. اطلاعات بیشتر در مورد پارامترها در فصل ۷ موجود است.

چک کردن و پاک کردن فایل های لاگ : لاگ فایل ها در طول زمان جمع می شوند، که ممکن است مشکل زا بشوند اگر شما از رآس برای مدتی روی سیستمی که محدودیت دارد (محدودیت فضای حافظه ای یا سخت افزاری). هر دو دستور roscore و roslaunch سایز لاگ موجود را چک می کنند، اگر بزرگتر از ۱ گیگ بایت GB بشود به شما هشدار می دهد، اما هیچ کاری برای کاهش سایز آن انجام نمی دهد. شما می توانید فضای حافظه ای که به وسیله کاربر فعلی برای لاگ رآس استفاده شده با دستور زیر چک کنید:[۱]

rosclean check

اگر لاگ های فضای زیادی را گرفته اند، شما می توانید همه ی لاگ فایل ها را با دستور زیر حذف کنید:

rosclean purge

همچنین می توانید لاگ فایل ها را دستی پاک کنید.

۴-۵- فعال کردن و غیرفعال کردن پیام های لاگ

اگر برنامه ی لیست ۴,۱، ۴,۴، و ۴,۵ را اجرا کنید (یا خروجی لیست ۴,۲ را با دقت بخوانید) ممکن متوجه شوید که هیچ پیامی در سطح دیباگ ایجاد نشده است، هرچند این این برنامه ها ماکرو ROS_DEBUG_STREAM را فراخوانده اند. چه اتفاقی برای این پیام های سطح دیباگ افتاده

[۱] http://wiki.ros.org/rosclean

است؟ جواب این است که، رآس به طور پیش فرض پیام ها با سطح INFO و بالاتر را ایجاد می کند و تلاش برای ایجاد کردن پیام دیباگ نادیده گرفته می شود.
این مثال، مثال خاصی از مفهوم سطح های لاگر (logger levels) است، که برای هر نود یک سطح شدت حداقل تعریف می کند. سطح لاگر به صورت پیش فرض INFO است، که دلیل نبود پیام در سطح دیباگ را در برنامه ی ما رو شن می کند. ایده ی سطحِ لاگر، امکان م شخص کردن سطح جزئیات برای هر لاگ نود، در زمان اجرای نود، را فراهم می کند.

تنظیم سطح لاگر مشابه فیلتر کردن سطح شدت در rqt_console است. تفاوتشان این است که تغییر در سطح لاگر از ایجاد پیام های لاگ از سورس خودشان برای نمایش جلوگیری می کند در حالیکه rqt_console هر پیام ورودی لاگ را می تواند فیلتر کند. به جز بعضی از سربرگ ها که می تواند تأثیر مشابهی داشته باشند.

برای پیام های لاگ که با سطح لاگر نشان داده شده اند، متن پیام حتی ارزیابی نمی شود. این مورد امکان پذیر است چون ROS_INFO_STREAM و ساختارهای مشابه فراخوان ماکرو هستند و نه تابع. اجرای این ماکروها فعال بودن پیام را چک می کند، و تنها زمانی متن پیام را ارزیابی می کند که ماکرو فعال باشد. این یعنی الف) شما نباید وابسته به تأثیرات جانبی احتمالی ناشی از ایجاد کردن رشته پیام باشید. و ب) غیرفعال کردن پیام های لاگ برنامه ی شما را کند نمی کند، حتی اگر پارامترِ ماکروِ لاگینگ برای ارزیابی زمانبر باشد.

روش های زیادی برای تنظیم سطح لاگر نود وجود دارد.

تنظیم سطح لاگر از کامند لاین (ترمینال): برای تنظیم کردن سطح لاگر نود از دستور زیر استفاده کنید:

rosservice call /node-name/set_logger_level ros.package-name level

این دستور سرویسی به نام set_logger_level را فرا می خواند، که به صورت خودکار به وسیله ی هر نود به وجود می آید. (در فصل ۸ سرویس ها را بیشتر مورد مطالعه قرار می دهیم.)

- node-name اسم نودی است که می خواهید سطح لاگرش را تنظیم کنید.
- package-name اسم پکیجی است که نود به آن تعلق دارد.
- پارامتر level رشته ای از بین FATAL، ERROR،WARN، INFO،DEBUG است که سطح لاگر را برای نود مشخص می کند.

برای مثال، برای فعال کردن پیام سطح دیباگ در مثال بالا، ما از دستور زیر استفاده می کنیم:

rosservice call /count_and_log/set_logger_level ros.agitr DEBUG

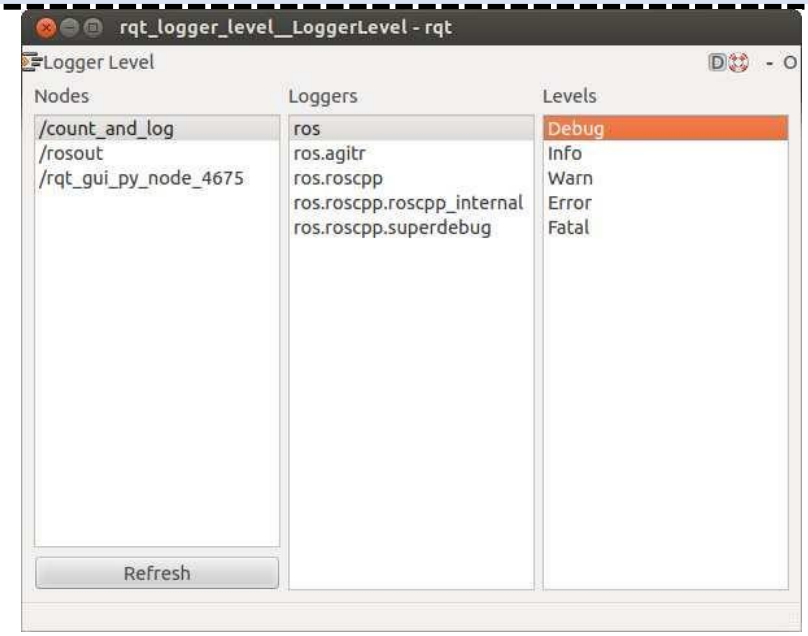
توجه داشته باشید که چون این دستور دقیقاً با خود نود ارتباط برقرار می کند، شما نمی توانید از دستور قبل از شروع نود استفاده کنید. اگر همه چیز درست کار کند خروجی فراخوانی rosservice یک خط خالی خواهد بود.

اگر شما سطح لاگر مورد نظر را اشتباه بنویسید، سرویس set_logger_level خطایی را گزارش می دهد، اما اگر قسمت ros.package-name اشتباه بنویسید خطایی نمی گیرد.

به آرگومان ros.package-name در rosservice برای مشخص شدن نام logger مورد نظر نیاز داریم. رآس به صورت داخلی کتابخانه ای به نام log4cxx را برای پیاده سازی این ویژگی لاگینگ استفاده می کند. هر آنچه در این فصل در موردش بحث شد، در پشت صحنه، از لاگر پیش فرض استفاده می کند که نامش ros.package-name است.

به هر حال، کتابخانه ی مشتری(Client) C++ در رآس نیز از لاگرهای داخلی دیگری استفاده می کند، برای دنبال کردن آنچه معمولاً برای کاربر جذاب نیست، در سطح پایین که بایتها خوانده و نوشته می شوند، ارتباطات برقرار یا قطع می شوند، callback ها خوانده می شوند. چون سرویس set_logger_level ارتباط با همه ی این لاگر ها را فراهم می کند، ما باید مشخص کنیم چه لاگری را می خواهیم تنظیم کنیم.

دلیل این سطح از پیچیدگی این است که دستور rosservice بالا اگر نام لاگر را اشتباه بنویسید خطا نمی دهد. به جای ایجاد خطا؛ log4cxx بی صدا یک لاگر جدید با نام مشخص را ایجاد می کند.



شکل (۴-۲) GUI برای rqt_logger_level.

تنظیم سطح لاگر از GUI: اگر شما یک GUI را به جای یک دستور در ترمینال ترجیح می دهید، از دستور زیر استفاده کنید:

```
rqt_logger_level
```

پنجره ی نتیجه که در شکل ۴,۲ نشان داده شده است، به شما اجازه می دهد که از لیست نود، لیست لاگر —شما حتماً به ros.package-name نیاز دارید— و لیست سطح لاگرها انتخاب کنید. تغییر سطح لاگر با استفاده از این ابزار همان تأثیر دستور rosservice را دارد، چون همان سرویس را برای هر نود فرا می خواند.

تنظیم سطح لاگر از کد C++: این امکان برای هر نود وجود دارد که سطح لاگرش را تغییر دهد. مستقیم ترین راه برای انجام اینکار دسترسی به زیر ساخت log4cxx است که رآس برای پیاده سازی ویژگی های لاگینگ از آن استفاده می کند. برای این کار از کد زیر می توان استفاده نمود.

```
#include <log4cxx/logger.h>
...
log4cxx::Logger::getLogger(ROSCONSOLE_DEFAULT_NAME)->setLevel(
ros::console::g_level_lookup[ros::console::levels::Debug]
);
ros::console::notifyLoggerLevelsChanged();
```

گذشته از پوشیدگی کد که ضروری می باشد، این کد باید به سادگی به عنوان تنظیمات سطح لاگر برای DEBUG شناخته شود. مسلماً عبارت DEBUG می تواند با Info ,Warn , Error , Fatal نیز جایگزین شود.

فراخواندن ros::console::notifyLoggerLevelsChanged() ضروری است چون حالت فعال/غیرفعال (enabled/disabled) هر لاگینگ ذخیره می شود. اگر شما سطح لاگر را قبل از هر لاگینگ تنظیم کنید، حالت قبل حذف می شود.

۴-۶- در ادامه

در این فصل دیدیم چگونه پیام های لاگ در برنامه رآس ایجاد کنیم، چگونه این پیام ها را به روش های مختلف ببینیم. این پیام ها برای دیباگ کردن و دنبال کردن رفتار پیچیده سیستم رآس مفید هستند، بخصوص وقتی این سیستم ها نودهای مختلفی را اجرا می کنند. در فصل بعدی در مورد اسم های رآس بحث می کنیم، که اگر هوشمندانه به کار ببریم، برای تشکیل سیستم پیچیده ای از نودهایی با قسمت های کوچکتر به ما کمک می کند.

فـصل ۵ : گرافِ اسم های منابع

در این فصل ما یاد می گیریم چگونه رآس اسم های نودها، تاپیک ها، پارامترها، و سرویس ها را استفاده می کند.

در فصل ۳، ما رشته هایی مانند "hello_ros" و "publish_velocity" برای نام نودها و رشته هایی مانند "turtle1/cmd_vel" و "turtle1/pose" برای نام تاپیک ها استفاده کردیم. همه ی این مثال ها نام های منابع گراف (graph resource names) هستند. رآس سیستم نام گذاری منعطفی دارد که انواع مختلف اسم ها را می پذیرد. (برای مثال این چهار مورد بالا اسم های نسبی هستند.) در این فصل، ما کمی از بحث منحرف می شویم تا انواع مختلف نام های منابع گراف را درک کنیم، و بفهمیم چگونه رآس آنها را استفاده می کند. ما این ایده ها ی بسیار ساده را در یک فصل جداگانه بیان می کنیم چون بسیاری از این مفاهیم به نیمه دوم کتاب مربوط هستند.

۵-۱- نام های جهانی (Global names)

نودها، تاپیک ها، سرویس ها، و پارامترها همه نوعی منابع گراف هستند. هر منبع گراف با یک رشته ی کوتاه که **نام منبع گراف** خوانده می شود مشخص می شوند.[۱] نام های منابع گراف همه جا هستند، در کامند لاین و کد رآس. هر دوی rosnode info و ros::init به نام نودها نیاز دارند؛ rostopic echo و ساختار ros::Publisher نام های تاپیک را نیاز دارند. همه ی این نمونه ها نام های منابع گراف هستند. در ادامه بعضی از نام های منابع گراف که تا کنون استفاده کرده ایم را می بینیم:

```
/teleop_turtle
/turtlesim
/turtle1/cmd_vel
/turtle1/pose
/run_id
/count_and_log/set_logger_level
```

این نام ها مثال هایی از کلاس مشخصی به نام global names هستند. این نام ها global names خوانده می شوند چون هرجایی می توانند استفاده شوند. این نام ها معنی روشن و مشخصی دارند، چه به عنوان آرگومانی از ابزارهای کامند لاین یا درون نودها استفاده شوند. برای فهمیدن منبعی که اسم بدان اشاره می کند، به هیچ اطلاعات دیگری نیاز نیست.

global name از قسمت های مختلفی ساخته شده است:

- پیشوند اسلش /، اسم را به عنوان اسم جهانی global مشخص می کند.

[۱] http://wiki.ros.org/Names

- توالی صفرها یا فضاهای نامی (namespaces)، به وسیله ی اسلش / از هم جدا می شوند. فضاهای نامی برای گروه کردن منابعِ گرافِ مربوط به هم استفاده می شود. نام های مثال بالا دارای دو فضای نامی turtle۱ و count_and_log هستند. می توان از چند سطح مختلف از فضای نامی استفاده کرد، بنابراین مثال زیر هم یک اسم جهانی global name است (گرچه نادر است)، و دارای ۱۱ فضای نامی تو در تو است.

/a/b/c/d/e/f/g/h/i/j/k/l

اسم های جهانی که به فضای نامی صریحاً اشاره نمی کنند (شامل سه تا از مثال های بالا) در فضای نامی جهانی global namespace هستند.

- یک نام اساسی base name نامی است که منابع خودش را تعریف می کند. نام های اساسی در مثال بالا turtlesim،teleop_turtle ، run_id، pose، cmd_vel، set_logger_level هستند.

توجه داشته باشید اگر نام های جهانی همه جا استفاده می شدند، پیچیدگی کمتری در استفاده از فضاهای نامی وجود داشت، بعلاوه اینکه احتمالاً دنبال کردن این موارد را برای انسان ساده تر می کرد. فایده ی اصلی این سیستم نامی از نام های نسبی و خصوصی نشأت می گیرد.

۵-۲- اسم های نسبی

اصلی ترین جایگزین برای اسم های جهانی، که در بالا ذکر شد، مشخصات کاملی از فضای نامی متشکل از نام ها ست که به رآس اجازه می دهد یک فضای نامی پیش فرض ایجاد کند. نام هایی که این ویژگی را استفاده می کنند اسم های منابع گراف نسبی relative graph resource Name، یا برای سادگی اسم نسبی relative name نامیده می شوند. مشخصه ی یک اسم نسبی عدم وجود پیشوند اسلش / در ابتدای نام است. مثال هایی از اسم های نسبی:

teleop_turtle
turtlesim
cmd_vel
turtle1/pose
run_id
count_and_log/set_logger_level

کلید اصلی فهم اسم های نسبی این است که به خاطر بسپاریم که اسم های نسبی نمی توانند به منابع گراف خاصی منطبق شوند مگر اینکه ما فضای نامی پیش فرض مورد استفاده ی رآس را بدانیم.

استفاده از اسم های نسبی : پروسه ی نگاشت اسم های نسبی به اسم های جهانی در واقع بسیار ساده است. برای استفاده از یک اسم نسبی به عنوان اسم جهانی، رآس نام فضای نامی فعلی را به اول اسم نسبی وصل میکند. برای مثال، اگر ما از اسم نسبی cmd_vel در جایی که فضای نامی پیش فرض /turtle1 است، استفاده کنیم، رآس نامی از ترکیب این دو را ایجاد می کند:

/turtle1 + cmd_vel ⇒ /turtle1/cmd_vel

فضای نامی پیش فرض — اسم نسبی — اسم جهانی

اسم نسبی می تواند با توالی فضاهای نامی شروع شود، که به صورت فضای تو در توی فضای نامی پیش فرض اعمال می شوند. به عنوان مثال، اگر شما از فضای نسبی g/h/i/j/k/l در جای که فضای نامی پیش فرض /a/b/c/d/e/f است، استفاده کنید، رآس ترکیب زیر را ایجاد می کند:

/a/b/c/d/e/f + g/h/i/j/k/l ⇒ /a/b/c/d/e/f/g/h/i/j/k/l

فضای نامی پیش فرض — اسم نسبی — اسم جهانی

و بعد نتیجه ی اسم جهانی برای مشخص کردن یک منبع گراف خاص استفاده می شود، دقیقاً همان طور که یک اسم جهانی از اول مشخص می کرد.

تنظیمات فضای نامی پیش فرض: فضای نامی پیش فرض به صورت جداگانه برای هر نودی دنبال می شود، به جای تنظیم یک سیستم گسترده. اگر شما هیچ قدم مشخصی برای تنظیم فضای نامی پیش فرض برندارید، همان طور که انتظار می رود رآس فضای نامی جهانی (/) را استفاده می کند. بهترین و متداول ترین روش برای انتخاب فضای نامی پیش فرض متفاوت برای یک نود یا گروهی از نودها استفاده از ns در لانچ فایل است. (بخش ۶,۳ را ببینید.) گرچه مکانیزم های مختلفی برای دستی انجام دادن وجود دارد.

- بیشتر برنامه های رآس، شامل همه ی برنامه های C++ که ros::init فرا می خوانند، پارامتری به نام __ns از کامند لاین می پذیرند، که نام فضای نامی پیش فرض را برای برنامه مشخص می کند.

__ns:=default-namespace

- شما همچنین می توانید نام فضای نامی پیش فرض را برای هر برنامه اجرا شده ی رآس داخل یک ترمینال، با استفاده از متغیر محیطی تنظیم کنید.

export ROS_NAMESPACE=default-namespace

این متغیر محیطی تنها در صورتی استفاده می شود که فضای نامی پیش فرض دیگری با پارامتر ns__ مشخص نشده باشد.

درک هدف اسم های نسبی: گذشته از این سؤال که چگونه فضای کاری پیش فرض اسم نسبی را مشخص کنیم، سؤال دیگر این است که " خوب که چی؟". در اولین نگاه، هدف از اسم های نسبی راه میانبری است برای اینکه هر دفعه اسم های جهانی را به طور کامل تایپ نکنیم. گرچه اسم های نسبی این راحتی را هم فراهم می کنند، هدف اصلی آنها راحتتر کردن ایجاد سیستم های پیچیده متشکل از قسمت های کوچکتر است.

وقتی یک نود از اسم های نسبی استفاده می کند، این توانایی را برای کاربر ایجاد می کند که به راحتی نود و تاپیک هایی که استفاده می کند را در زیر فضای نامی که برای طراح مشخص نبوده استفاده کند. این نود انعطاف تشکیل یک سیستم را مشخص تر می کند و مهمتر از آن، زمانی که گروهی از نودها از منابع مختلف ترکیب می شوند می تواند از تداخل اسم ها جلوگیری کند. از سویی دیگر هر اسم جهانی صریح رسیدن به این ترکیب را سختتر می کند. بنابراین، وقتی نودی را می نویسید، پیشنهاد می شود از اسم های جهانی استفاده نکنید، بجز مواقع استثنائی که دلیل خوبی برای استفاده از آنها دارید.

۵-۳- اسم های خصوصی

اسم های خصوصی، Private names، که باید یک مد(~) شروع می شوند، سومین و آخرین کلاس نامهای منابع گراف هستند. مانند اسم های نسبی، اسم های خصوصی نیز فضای نامی که در آن هستند را به صورت کامل مشخص نمی کنند و وابسته به کتابخانه های مشتری رآس برای ایجاد یک اسم جهانی کامل هستند. تفاوت این است که به جای استفاده از فضای نامی پیش فرض، اسم های خصوصی از اسم نودشان به عنوان فضای نامی استفاده می کنند.
برای مثال، برای نودی که اسم جهانی اش /sim1/pubvel است، اسم خصوصی ~max_vel به اسم جهانی مشابه زیر تبدیل می شود:

$$\underbrace{\text{/sim1/pubvel}}_{\text{اسم نود}} + \underbrace{\text{~max\_vel}}_{\text{خصوصی اسم}} \Rightarrow \underbrace{\text{/sim1/pubvel/max\_vel}}_{\text{اسم جهانی}}$$

دلیلش این است که هر نود فضای نامی خودش را دارد برای چیزهایی که فقط به همان نود مربوط هستند، و برای هیچ چیز دیگری استفاده ایی ندارند. اسم های خصوصی معمولاً برای پارامترها

استفاده می شوند، roslaunch یک ویژگی مشخصی برای تنظیم پارامترهایی دارد که به وسیله ی اسم های خصوصی قابل دسترس هستند. در صفحه ۱۱۳، سرویس هایی که عملکرد نودی را کنترل می کند را ببینید. معمولاً استفاده از یک نام خصوصی برای ارجاع به یک تاپیک اشتباه است، اگر ما می خواهیم نودها به هم وابسته نباشند، هیچ تاپیکی نباید به نود خاصی تعلق داشته باشد.

اسم های خصوصی فقط با این مفهوم خصوصی هستند که آنها در فضای نامی هستند که قرار نیست به وسیله ی نودهای دیگر استفاده شود. منابع گراف ارجاع داده شده به وسیله ی اسم های خصوصی به وسیله ی اسم های جهانیشان برای هر نودی که اسمشان را بداند قابل دسترس هستند. این در تضاد با کلید واژه ی خصوصی private در C++ و زبان های برنامه نویسی مشابه است، که از دسترسی بقیه قسمت های سیستم به اعضای مشخصی از کلاس جلوگیری می کند.

۵-۴- اسم های مستعار

بعلاوه ی این سه نوع اصلی اسم ها، رآس نوع دیگری مکانیزم اسم گذاری به نام اسم های مستعار anonymous names دارد، که مخصوص اسم نودها استفاده می شود. هدف از اسم های مستعار اعمال قانون تک بودن اسم هر نود است. ایده این است که هر نود می تواند، در حین فراخوانی ros::init، درخواست بدهد به صورت خودکار به یک اسم تک نسبت داده شود.
برای درخواست یک اسم مستعار، یک نود باید ros::init_options::AnonymousName را به عنوان چهارمین پارامتر ros::init وارد کند:

```
ros::init(argc, argv, base_name, ros::init_options::AnonymousName);
```

تأثیر این گزینه ی اضافه این است که پسوند هایی به اسم اولیه اضافه می کند، تا مطمئن شود اسم نود یکتا است.

گرچه جزئیات پسوندی که اضافه می شود مهم نیست، اما جالب است که بدانید ros::init زمان فعلی را برای تشکیل اسم مستعار استفاده می کند.

```
1   // This program starts with an anonymous name, which
2   // allows multiple copies to execute at the same time,
3   // without needing to manually create distinct names
4   // for each of them.
5   #include <ros/ros.h>
6   int main(int argc, char **argv) {
7     ros::init(argc, argv, "anon",
8       ros::init_options::AnonymousName);
9     ros::NodeHandle nh;
10    ros::Rate rate(1);
11    while(ros::ok()) {
12      ROS_INFO_STREAM("This message is from "
13       << ros::this_node::getName());
14      rate.sleep();
15    }
16  }
```

لیست (۵-۱) برنامه ی anon.cpp که نودهایش اسم مستعار دارند. می توان تعداد زیادی از کپی های این برنامه را به طور همزمان اجرا کنیم، بدون اینکه اسم نودها با هم تداخلی داشته باشند.

لیســـت ۵,۱ یک برنامه را که از این ویژگی اســـتفاده می کند را نشـــان می دهد. به جای اینکه به سادگی anon نام بگیرد، نودهایی که به وسیله ی این برنامه شروع می شوند به صورت زیر نام می گیرند:

/anon_1376942789079547655
/anon_1376942789079550387
/anon_1376942789080356882

رفتار برنامه کاملاً عادی است، اما چون درخواست یک اسم مستعار کرده است، ما می توانیم تعداد بی شماری از این برنامه را به صورت همزمان اجرا کنیم، با توجه به اینکه به هر کدام از آنها هنگام شروع یک اسم یکتا نسبت داده می شود.

۵-۵- در ادامه

در این فصل دیدیم که رآس چگونه منابع گراف را تبدیل می کند. به خصوص، سیستم های بزرگ رآس با تعداد زیادی از نودهای مرتبط می توانند از این انعطاف ناشی از اسم های نسبی و خصوصی سود ببرند (به خوبی ا ستفاده کنند). در فصل بعد ابزاری به نام roslaunch را معرفی می کنیم که شروع و تنظیم پروسه های شامل چند نود رآس را ساده می کند.

فـصل ۶ : فایل های لانچ

در این فصل ما یاد می گیریم چند نود را با هم به وسیله ی لانچ فایل تنظیم و اجرا کنیم.

اگه شـما با همه مثال ها تا اینجا کار کرده باشـید، باید از اجرای دسـتی تعداد زیادی نود مختلف، تازه بدون roscore، در ترمینال های مختلف خسته شده باشید. خوشبختانه، رآس ماکانیزمی برای شـروع همزمان مسـتر و تعداد زیادی نود به وسـیله ی فایلی به نام فایل لانچ launch file فراهم کرده است. استفاده از فایل های لانچ بین پکیج های زیادی گسترش دارد. هر سیستمی که بیشتر از یک یا دو نود اسـتفاده می کند از مزایای فایل لانچ برای مشـخص کردن و تنظیم کردن نودها استفاده می کند. این فصل این فایل ها و ابزار roslaunch که از آنها استفاده می کند را معرفی می کند.

۶-۱- استفاده از فایل های لانچ

بیایید ببینیم چگونه فایل لانچ این امکان را برای ما فراهم می کند که نودهای زیادی را همزمان اجرا کنیم. ایده اولیه لیسـت کردن گروهی از نودها که باید همزمان اجرا شـوند، درون یک فرمت XML مشـخص، اسـت.[۱] لیسـت ۶,۱ مثال کوچکی از فایل لانچ را نشـان می دهد که شـبیه سـاز turtlesim به همراه نود کنترل از راه دور که در فصـل دو دیدیم و نود شـنوده که در فصـل سـوم نوشتیم را شروع می کند. این فایل به اسم example.launch در پوشه ی اصلی پکیج agitr ذخیره شده ا ست. قبل از اینکه وارد جزئیات فایل لانچ ب شویم، بیایید نحوه ی ا ستفاده از این فایل ها را ببینیم.

[۱] http://wiki.ros.org/roslaunch/XML

```
1   <launch>
2    <node
3     pkg="turtlesim"
4     type="turtlesim_node"
5     name="turtlesim"
6     respawn="true"
7    />
8    <node
9     pkg="turtlesim"
10    type="turtle_teleop_key"
11    name="teleop_key"
12    required="true"
13    launch-prefix="xterm -e"
14   />
15   <node
16    pkg="agitr"
17    type="subpose"
18    name="pose_subscriber"
19    output="screen"
20   />
21  </launch>
```

لیست (۶-۱) فایل لانچ example.launch که سه نود را همزمان اجرا می کند.

اجرای فایل های لانچ: برای اجرای فایل لانچ، دستور roslaunch را استفاده کنید:[۱]

roslaunch package-name launch-file-name

شما می توانید فایل لانچ مثال را بدین صورت فرا بخوانید:

roslaunch agitr example.launch

اگر همه چیز به درستی کار کند، این دستور سه نود را اجرا می کند. شما باید پنجره ی turtlesim، به همراه پنجره ی دیگری که ورودی کلید های برداری برای کنترل از راه دور لاک پشـــت می پذیرد، را ببینید. ترمینال اصـلی که شـما دسـتور roslaunch را در آن اجرا کرده اید باید اطلاعات موقعیت را نشان دهد که به وسیله ی برنامه ی subpose وارد شده است. قبل از شروع هر نودی، roslaunch تشخیص می دهد که آیا roscore اجرا شده است یا نه، اگر نه، به صورت خودکار آن را اجرا می کند.

مواظب باشـــید rosrun که تنها یک نود را اجرا می کند، را با roslaunch، که چندین نود را همزمان اجرا می کند، اشتباه نگیرید.

[۱] http://wiki.ros.org/roslaunch/CommandlineToolshttp://wiki.ros.org/roslaunch/CommandlineTools

این امکان نیز وجود دارد که یک فایل لانچ که به هیچ پکیجی تعلق ندارد را اجرا کنید. برای این کار به roslaunch مســـیر فایل لانچ را بدهید، بدون اشـــاره به هیچ پکیجی. برای مثال، در کامپیوتر نویسنده، این دستور مثال فایل لانچ را اجرا می کند، بدون در نظر گرفتن این قضیه که این فایل جزئی از یک پکیج رآس است:

roslaunch ~ /ros/src/agitr/example.launch

این نوع دور زدن تشکیلات معمول پکیج ها اصلاً ایده ی خوبی نیست، و فقط برای یک آزمایش خیلی ساده و کوتاه جایز است.

یک قضـــیه ی مهم در مورد roslaunch (که ممکن اســـت فراموش شـــود) این اســـت که همه ی نودهای یک فایل لانچ تقریباً در یک زمان شـــروع می شـــوند. در نتیجه شـــما نمی توانید مطمئن باشـــید کدام نود آنها را مقدار دهی اولیه می کند. نودهای درســـت نوشـــته شـــده به ترتیب اجرای خودشـــان و وابســـتگانشـــان اهمیتی نمی دهند. (بخش ۷,۳ مثالی را نشـــان می دهد که ترتیب اجرا مهم است.)

این رفتار فلســـفه ی رآس را مشـــخص می کند که هر نود باید از نودهای دیگر مســـتقل باشـــد. (بحث ما در بخش ۲-۸ در مورد اســـتقلال نودها را بیاد بیاورید.) نودهایی که فقط در ترتیب خاصی اجرا می شوند، برای این طراحی ماژولار مناسب نیستند. چنین نودهایی اغلباً می توانند دوباره طوری طراحی شوند که این محدودیت ترتیب را نداشته باشند.

درخواست اضافه : مانند همه ی ابزارهای کامند لاین، roslaunch هم گزینه ای برای درخواست اضافه دارد:

roslaunch -v package-name launch-file-name

لیست ۶,۲ مثالی از اطلاعات را نشان می دهد که این گزینه علاوه بر پیام های حالت معمول ایجاد می کند. این گزینه می تواند در مواردی برای اشـــکال زدایی مفید باشـــد تا جزئیاتی را ببنید که roslaunch چگونه فایل لانچ شما را ترجمه می کند.

```
1  ... loading XML file [ /opt/ros/indigo/etc/ros/roscore.xml]
2  … executing command param [ rosversion roslaunch]
3  Added parameter [ /rosversion]
4  … executing command param [ rosversion−d]
5  Added parameter [/rosdistro]
6  Added core node of type [rosout/rosout] in namespace [/]
7  … loading XML file [/home/jokane/ros/agitr/example.launch]
8  Added node of type [turtlesim/turtlesim_node] in namespace [/]
9  Added node of type [agitr/pubvel] in namespace [/]
10 Added node of type [agitr/subpose] in namespace [/]
```

لیست (۶-۲) خروجی اضافی که به وسیله مد اضافه ی roslaunch ایجاد شده است.

متوقف کردن فایل لانچ: برای توقف یک roslaunch فعال، می توانید از Ctrl-C اســتفاده کنید. این ســیگنال تلاش می کند تدریجاً همه ی نودهای فعال فایل لانچ را خاموش کند، و هر نودی را که بعد از مدت کوتاهی خاموش نشود را با اجبار می کشد.

۶-۲- ساختن فایل لانچ

دیدیم که چگونه می توان از یک فایل لانچ اســتفاده کرد، اکنون ما آماده هســتیم در مورد اینکه چگونه آنها را برای خودمان بسازیم فکر کنیم.

۶-۲-۱- کجا باید فایل ها ی لانچ را قرار دهیم

مانند همه ی فایل های دیگر رآس، هر فایل لانچ باید به یک پکیج خاص نســبت داده شــود. نحوه ا سم گذاری معمول برای ا سم های فایل های لانچ ا ستفاده از launch. در پایان ا سم ا ست. ساده ترین مکان برای ذخیره ی فایل های لانچ پوشــه ی پکیج اســت. وقتی دنبال فایل لانچ می گردیم، roslaunch زیرپوشه های هر پوشه ی پکیج را نیز جستجو می کند. بعضی از پکیج ها، شامل پکیج های اصــلی رآس، از این ویژگی با قرار دادن فایل های لانچ درون زیرپوشــه ای به نام launch استفاده کرده اند.

۶-۲-۲- مواد اولیه

ساده ترین فایل لانچ شامل یک المان اصلی در برگیرنده ی المان های نودهای مختلف است.

وارد کردن المان ا صلی: فایل های لانچ اسناد XML هستند، هر سند XML باید دقیقاً یک المان اصلی داشته باشد.برای فایل های لانچ رآس، المان اصلی با یک جفت برچسب launch تعریف شده است:

```
<launch>
...
</launch>
```

بقیه المان های دیگر فایل لانچ باید درون این برچسب ها قرار گیرند.

لانچ کردن نودها: قلب هر فایلِ لانچ یک ترکیب از المان های نود است، که هر کدام اسم یک نود را برای لانچ شدن است.[۱] یک المان نود مشابه زیر است:

```
<node
pkg="package-name"
type="executable-name"
name="node-name"
/>
```

اسلش قبلی در آخر برچسب node بسیار مهم است و به راحتی فراموش می شود. این را نشان می دهد که هیچ تگی برای خاتمه </node> نخواهد آمد، آن المان node کامل است. تجزیه کننده های XML باید در این قضایا بسیار سخت گیر باشند. اگر شما این اسلش / را حذف کنید، برای این نوع خطاها آماده باشید:

Invalid roslaunch XML syntax: mismatched tag

شما می توانید همچنین برچسب خاتمه را مشخصاً بنویسید:

```
<node pkg=". . . " type=". . . " name=". . . "></node>
```

در واقع اگر نود دارای زیر شاخه باشد ما به این تگ خاتمه ی جدا نیاز داریم، زیر شاخه ای مانند المان های remap و param. این المان ها در بخش ۶,۴ و ۷,۴ به ترتیب معرفی شده اند.

یک المان نود نیاز به سه ویژگی دارد:

[۱] http://wiki.ros.org/roslaunch/XML/node

- ویژگی های pkg و type مشخص می کنند که چه برنامه ای از رآس باید اجرا شود تا نود اجرا شـود. اینها مانند آرگومان های کامند لاین rosrun هسـتند،که به ترتیب نام پکیج و نام قابل اجرا را مشخص می کنند.
- ویژگی name نامی را به نود نسبت می دهد. این مورد هر نامی که نود به صورت معمول با استفاده از فراخوانی ros::init به خودش نسبت داده باشد را بازنویسی می کند.

این بازنویسی همه ی اطلاعات نامی که به وسیله ros::init ایجاد شده را از بین می برد، شامل هر درخواستی که نود احتمالاً برای ایجاد اسم مستعار دارد. (بخش ۵,۴ را ببنید.) برای استفاده از اسم مستعار در فایل لانچ، از یک جایگزین anon[1] برای ویژگی name استفاده کنید، مانند زیر:

name="$(anon base_name)"

به هر حال توجه داشته باشید استفاده چندباره از base_name اسم های مستعار مشابه ای را می سازد. این یعنی الف) ما می توانیم به آن نام در قسمت های دیگر فایل لانچ ارجاع بدهیم، اما ب) ما باید مواظب استفاده از base_name های مختلف هر نود که می خواهیم اسم مستعار داشته باشد باشیم.

پیدا کردن نود فایل های لاگ: مهمترین تفاوت بین roslaunch و اجرای جداگانه ی هر نود به وسیله ی rosrun است که، به صورت پیش فرض، خروجی استاندارد نودهای لانچ شده در یک فایل لاگ نوشـته می شـوند، و در کنسـول نشـان داده نمی شـوند.(۱- در ویرایش فعلی roslaunch، خروجی خطای استاندارد ـ خروجی کنسول شامل پیام های لاگ سطح ERROR و FATAL،علاوه بر فایل لاگ در کنسول هم ظاهر می شوند. به هر حال، یک نو شته ای در کد سورس roslaunch بدین مو ضوع ا شاره کرده ا ست که این رفتار ممکن ا ست در آینده تغییر کند.) نام فایل لاگ این است:

~ /.ros/log/run_id/node_name-number-stdout.log

Run_id یک شنا سه ی یکتا ا ست که زمان شروع م ستر ایجاد می شود. (برای جزیئات در مورد اینکه چگونه run_id فعلی را پیدا کنیم به صـفحه ۶۹ مراجعه کنید.) اعداد در اسـم این فایل ها اعداد صحیح کوچکی ه ستند که تعداد نود ها را می شمارند. برای مثال اجرای فایل لانچ لی ست ۶,۱ خروجی استاندارد دو نودش را به فایل لاگ بدین اسامی می فرستد:

turtlesim-1-stdout.log
Telep_key-3-stdout.log

فایل های لاگ می توانند به وسیله ی هر ویرایشگر متنی دیده شوند.

[1] http://wiki.ros.org/roslaunch/XML

فرســتاندن خروجی به کنســول : برای بازنویســی این رفتار برای هر نود، از ویژگی output در المان نودش استفاده کنید:

output="screen"

نودهایی که با این ویژگی لانچ شده اند خروجی استانداردشان را در صفحه نمایش نشان می دهند به جای اینکه به فایل لاگ بالا بفرستند. این مثال این ویژگی را برای نود subpose اسـتفاده کرده اسـت، که روشـن می کند چرا پیام های INFO از این نود در کنسـول ظاهر می شـوند. همچنین روشن می کند چرا این نود در لیست فایل های لاگ بالا حذف شده است.

علاوه بر ویژگی output، که روی یک نود تنها تأثیر می گذارد، ما همچنین می توانیم roslaunch را مجبور کنیم خروجی همه ی نودها را نشان دهد با استفاده از کامند لاین screen-- :

roslaunch --screen package-name launch-file-name

اگر یک برنامه که ازroslaunch شروع می شود، خروجی مورد نظر شما را نشان نداد، شما باید مطمئن شوید نود ویژگی "output="screen را دارد.

درخواست اجرای مجدد: بعد از شروع همه ی نودها، roslaunch همه نودها را دنبال می کند، تا بفهمد که کدام نودها فعال باقی مانده اند. برای هر نود، ما می توانیم از roslaunch بخواهیم اگر متوقف شد دوباره آن را اجرا کند، با استفاده از ویژگی respawn :

respawn="true"

این ویژگی برای مثال، برای نودهایی که به صورت موقت به دلیل مشکل نرم افزاری، سخت افزاری و یا هر دلیل دیگری متوقف می شوند، می تواند مفید باشد.

ویژگی respawn واقعاً در مثال ما لازم نیست (همه این سه برنامه کاملاً قابل اطمینان هستند) اما ما آن را در turtlesim_node بکاربردیم که رو شن کنیم نود چگونه مجدد اجرا می شود. اگر شما پنجره ی turtlesim را ببندید، نود مربوطه متوقف می شود. رآس سریعاً متوجه می شود (چون این نود به عنوان نود respawn م شخص شده ا ست) و یک نود جدید turtlesim، با پنجره ی مربوطه، جای قبلی ظاهر می شود.

درخوا ست نودها: یک جایگزین برای respawn این ا ست که اظهار کنیم یک نود لازم required است:

required="true"

وقتی یک نود required متوقف می شود، roslaunch با متوقف کردن همه ی نودهای فعال دیگر و خارج شـدن از خودش پاسـخ می دهد. این نوع رفتار ممکن اسـت مفید باشـد، برای مثال، برای نودهایی که الف) اگر از کار افتادند همه ی بخش ها باید ترک شـوند، ب) نمی توانند به آرامی با ویژگی respawn ریست شود.

این مثال ویژگی required را برای نود turtle_teleop_key اسـتفاده می کند. اگر شـما پنجره ای که نود کنترل از راه دور در آن اجرا شـده اسـت را ببندید، roslaunch دو نود دیگر را متوقف می کند و خارج می شود.

اگر هر دو ویژگی respawn و required برای یک نود تنظیم کنیم، چون مفاهیمشان با همدیگر همخوانی ندارند، roslaunch اعتراض می کند.

لانچ کردن نودها در پنجره ی خود شان: یک ا شکال بالقوه برای ا ستفاده از roslaunch، نسبت به روش اصلی استفاده از rosrun در یک ترمینال جداگانه برای هر نود، این است که همه ی نودها یک ترمینال را به اشـتراک می گذارند. این مورد برای نودهایی که پیام های لاگ ایجاد می کنند و ورودی کنسـول را قبول نمی کند، قابل کنترل (و اغلب مفید) اسـت. برای نودهایی که به ورودی کن سول وابـ سته ه ستند، به عنوان نمونه turtle_teleop_key، احتمالاً باقی ماندن در یک ترمینال جداگانه ترجیح دارد.

خوشبختانه، roslaunch راه تمیزی برای این مورد دارد، استفاده از ویژگی launch-prefix در المان نود:

launch-prefix="command-prefix"

بدین ترتیب که roslaunch پیـشوند داده شده را در ابتدای کامند لاینی که به صورت داخلی برای اجرا نود سـاخته اسـت قرار می دهد. در example.launch ما از این ویژگی برای نود کنترل از راه دور استفاده کردیم:

launch-prefix="xterm -e"

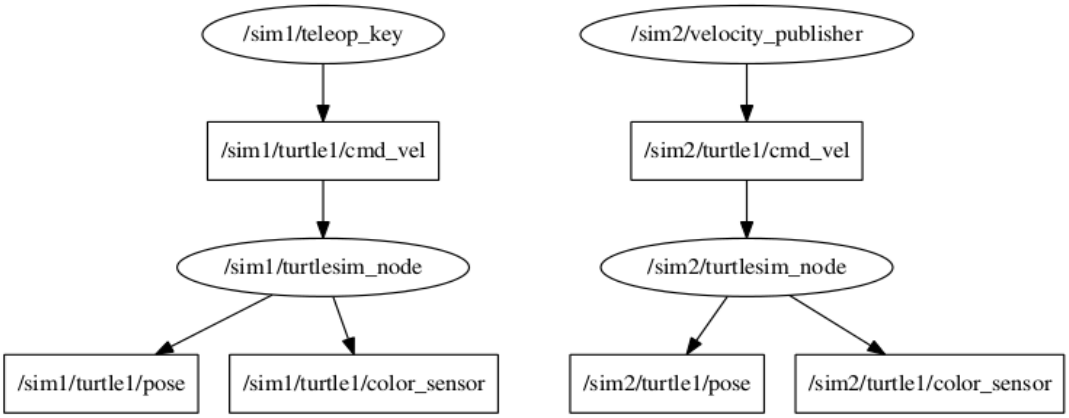
به خاطر این ویژگی، المان این نود تقریباً برابر دستور زیر است:

xterm -e rosrun turtlesim turtle_teleop_key

همان طور که می دانید، دسـتور xterm یک پنجره ی ترمینال جدید را شـروع می کند. آرگومان e- به xterm می گو ید که بق یه دسـتورات این خط را (در این مورد، rosrun turtlesim turtle_teleop_key) درون خودش اجرا کند، به جای ترمینال جد ید. نتیجه این اسـت که turtle_teleop_key، یک برنامه کاملاً وابسته به متن، درون یک پنجره ی گرافیکی ظاهر می شود.

ویژگی launch-prefix مطمئناً فقط وابسته به xterm نیست. این ویژگی می تواند برای ا شکال زدایی نیز مفید باشـد (با اسـتفاده ازgdb و valgrind)، یا کاهش اولویت زمانبندی پروسـه (با استفاده از nice[1])

[1] http://wiki.ros.org/rqt_console



شکل (۶-۱) نودها و تاپیک ها (به ترتیب در بیضی و مستطیل) که به وسیله ی doublesim.launch ایجاد شده اند.

۶,۳ لانچ کردن نودها درون یک فضای نامی

ما در بخش ۵,۲ دیدیم که رآس اسم های نسبی را پشتیبانی می کند، که مفهوم فضای نامی پیش فرض را ا ستفاده می کند. یک روش معمول برای تنظیم فضای نامی پیش فرض برای هر نود (یک پرو سه ای که اغلب pushing down به یک فضای نامی ا ست) ا ستفاده از یک فایل لانچ و نسبت دادن ویژگی ns در المان نودش است:

ns="namespace"

لیست ۶,۳ مثالی از فایل لانچ را نشان می دهد که از این ویژگی برای ساختن دو شبیه ساز turtlesim مستقل استفاده می کند. شکل ۶,۱ نودها و تاپیک های ناشی از این فایل لانچ را نشان می دهد.

- در turtlesim اسم های معمول تاپیک ها (turtle1/pose، turtle1/color_sensor، turtle1/cmd_vel) از فضای نامی جهانی به فضای نامی جداگانه به نام sim1/ و sim2/ منتقل می شوند. این تغییر به این دلیل اتفاق افتاد که کد turtlesim_node از اسم های نسبی مانند turtle1/pose در ایجاد ros::Publisher و ros::Subscriber ا ستفاده می کند (به جای استفاده از اسم های جهانی مانند /turtle1/pose).
- به همین ترتیب، اسم نودها در فایل لانچ اسم های نسبی هستند. در این مورد، هر دو نود اسم نسبی مشابهی دارند، turtlesim_node. این مشابهت اسم های نسبی مشکلی ندارد،

چون اســـم هـای جهـانی کـه برای آنهـا اســـتفـاده می شـــود متفـاوت اســـت (/sim1/turtlesim_node و /sim2/turtlesim_node).

- در واقع roslaunch به اسم نود به عنوان اسم پایه در فایل های لانچ نیاز دارد، اسم نود به صورت ا سم ن سبی بدون ا شاره به هیچ ف ضای نامی باید با شد، و در غیر صورت با ظاهر شدن اسم جهانی در ویژگی name در المان نود اخطار می دهد.

```
<launch>
 <node
  name="turtlesim_node"
  pkg="turtlesim"
  type="turtlesim_node"
  ns="sim1"
 />
 <node
  pkg="turtlesim"
        type="turtle_teleop_key"
        name="teleop_key"
        required="true"
        launch-prefix="xterm -e"
        ns="sim1"
       />
       <node
        name="turtlesim_node"
        pkg="turtlesim"
        type="turtlesim_node"
        ns="sim2"
       />
       <node
        pkg="agitr"
        type="pubvel"
        name="velocity_publisher"
        ns="sim2"
       />
      </launch>
```

لیست (۶-۳) یک فایل لانچ به نام doublesim.launch که دو نود مستقل turtlesim را شروع می کند. در یک شبیه ساز یک لاک پشت با دستور سرعتی به صورت تصادفی حرکت می کند و دیگری با کنترل از راه دور.

این مثال مشـابهت هایی با سـیسـتم بحث شـده در بخش ۲,۸ دارد. در هر دو مورد، ما چند نود turtlesim را شروع کردیم. اما نتایج با هم متفاوت است. در بخش ۲,۸، ما فقط اسم نودها را تغییر دادیم، و نودها همچنان در یک فضـای نامی باقی ماندند. در نتیجه، هر دو نود turtlesim به یک تاپیک گوش می دهند و یک تاپیک را منتشر می کنند. راه مستقیمی برای ارتباط با این دو شبیه سـاز به طور جداگانه وجود ندارد. در مثال جدید لیسـت ۶,۳، ما هر نود را در فضـای نامی خودش قرار می دهیم. در نتیجه ا سم تاپیک ها تغییر می کنند و دو شبیه ساز م ستقل از هم می شوند، این امکان را برای ما فراهم می کنند که دستورهای سرعت متفاوتی برای هر کدام منتشر کنیم.

در این مثال، فضای نامی که با ویژگی ns مشخص شده اند هم اسم نسبی هستند. ما از اسم های sim1 و sim2 در متن فایل لانچ استفاده کردیم در حالیکه فضای نامی پیش فرض، ف ضای نامی جهانی (با ا سلش /) ا ست. در نتیجه ف ضای نامی پیش فرض برای این دو نود sim1/ و sim2/ می شود.

به صـورت فنی این امکان وجود دارد که برای این ویژگی یک فضـای جهانی فراهم کنیم. هر چند، بی شتر اوقات این ایده ی بدی ا ست، به همان دلیلی که ا ستفاده از ا سم جهانی داخل نود ایده ی بدی ا ست. اینگونه در صورتیکه فایل لانچ داخل فایل لانچ دیگری فراخوانده شود، فایل لانچ اولی از فضای نامی خودش استفاده نمی کند.

۶-۳- نگاشت اسم ها

علاوه بر اسم های نسبی و خصوصی، نودهای رآس همچنین نگاشت ها را نیز پشتیبانی می کنند، که سطح بهتری از کنترل برای تغییر اسم های استفاده شده در نود را فراهم می کند.[۱] نگاشت ایده ایست براساس جانشین کردن: هر نگاشت جایگزینی اسم اصلی با اسم جدید است. هر دفعه که نود نگا شتی از ا سم های ا صلی اش را ا ستفاده می کند، کتابخانه ی م شتری (client) رآس در پ شت صحنه آن را با اسم جدید عوض می کند.

۶-۳-۱- ایجاد یک نگاشت

دو راه برای ایجاد نگاشت هنگام شروع نود وجود دارد.

[۱] http://wiki.ros.org/RemappingArguments

- برای نگاشت یک اسم هنگام شروع نود از کامند لاین، اسم اصلی را با اسم جدید، جدا شده به وسیله ی := در کامند لاین استفاده کنید.

original-name:=new-name

برای مثال، برای اجرای یک نمونه turtlesim که موقعیتش را روی تاپیکی به نام tim/ به جای turtle1/pose/ منتشر می کند، از کامند لاینی بدین صورت استفاده کنید:

rosrun turtlesim turtlesim_node turtle1/pose:=tim

- برای نگاشت اسم ها درون فایل لانچ، از المان remap استفاده کنید.[1]

<remap from="original-name" to="new-name" />

اگر در سطح بالا، به عنوان زیر شاخه ی المان launch، ظاهر شود این نگا شت در مورد همه ی نودهای بعدی اجرا می شـــود. المان remap می تواند به عنوان زیر شـــاخه ای از المان node نیز ظاهر شود، مانند:

```
<node node-attributes >
<remap from="original-name" to="new-name" />
...
</node>
```

در این مورد، نگاشـــت فقط برای یک نود که بدان تعلق دارد اجرا می شـــود. برای مثال، کامند لاین بالا دقیقاً برابر با ساختار فایل لانچ زیر است:

```
<node pkg="turtlesim" type="turtlesim_node"
name="turtlesim" >
<remap from="turtle1/pose" to="tim" />
</node>
```

یک مورد مهم که در مورد نگاشت باید به خاطر بسپاریم این است که: همه ی اسم ها، شامل اسم های اصلی و جدید، به اسم های جهانی تبدیل می شوند، قبل از اینکه رآس نگاشتی را انجام دهد. در نتیجه، اسم هایی که در نگا شت ظاهر می شوند معمولاً اسم نسبی هستند. بعد از کامل شدن اسم، نگاشت با یک تطابق رشته ای مستقیم انجام می شود، با جستجو اسم های استفاده شده در نود که دقیقاً مشابه اسم های اصلی نگاشت هستند.

۶-۳-۲- معکوس کردن یک لاک پشت

برای ساختن یک مثال برای اینکه بفهمیم چگونه این نوع نگاشت مفید خواهد بود، سناریویی را در نظر بگیرد که ما می خواهیم با استفاده از turtle_teleop_key لاک پشت turtlesim را با استفاده

[1] http://wiki.ros.org/roslaunch/XML/remap

از کلید های جهت کیبورد کنترل کنیم، اما به صـــورت برعکس. به این صـــورت که با اســـتفاده از کلیدهای جهت چپ و راست می خواهیم لاک پشت به ترتیب ساعتگرد و پادساعتگرد بچرخد، و با کلید های بالا و پایین لاک پ شت به ترتیب عقب و جلو برود. این مثال ممکن ا ست غیر معمول به نظر برسد، اما در واقع کلاس معمولی از مسائله ی واقعی است که پیام های منتشر شده ی یک نود باید در فرمت دیگری که نود دیگری انتظار دارد ترجمه شوند.

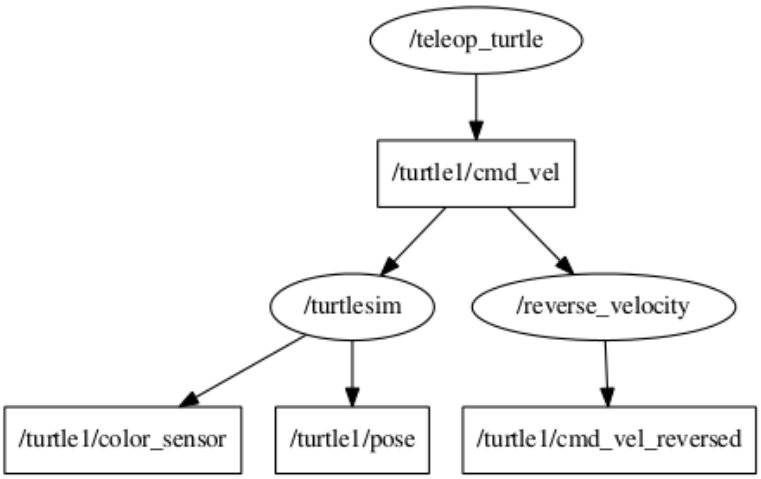
یک راه، این اســت که ســورس کد turtle_teleop_key را کپی کنیم و به صــورتی که می خواهیم تغییر دهیم. این گزی نه خیلی ناخوشـــایـ ند اســـت، چون این مورد ن یاز دارد که ما کد turtle_teleop_key را بفهمیم و از آن بدتر کد را تکرار کنیم. در عوض، ببنیم چگونه این کار را انجام دهیم، برنامه ی جدیدی ایجاد کنیم که کامند سرعت منتشر شده از نود کنترل از راه دور را معکوس می کند.

لیســت ۶,۴ یک برنامه ی کوتاه برای اجرای این تغییر نشــان می دهد: به turtle1/cmd_vel گوش می دهد و برای هر پیامی که دریافت می کند، هر دو دســـتور ســـرعت خطی و زاویه ای (linear و angular) را معکوس می کند، و نتیجه را روی turtle1/cmd_vel_reversed/ منتشر می کند.

```
1  // This program subscribes to turtle1/cmd_vel and
2  // republishes on turtle1/cmd_vel_reversed,
3  // with the signs inverted.
4  #include <ros/ros.h>
5  #include <geometry_msgs/Twist.h>
6  ros::Publisher *pubPtr;
7  void commandVelocityReceived(
8    const geometry_msgs::Twist& msgIn)
9  {
10           geometry_msgs::Twist msgOut;
11           msgOut.linear.x = -msgIn.linear.x;
12           msgOut.angular.z = -msgIn.angular.z;
13           pubPtr->publish(msgOut);
14         }
15         int main(int argc, char **argv) {
16           ros::init(argc, argv, "reverse_velocity");
17           ros::NodeHandle nh;
18           pubPtr = new ros::Publisher(
19             nh.advertise<geometry_msgs::Twist>(
20               "turtle1/cmd_vel_reversed",
21               1000));
22           ros::Subscriber sub = nh.subscribe(
23             "turtle1/cmd_vel", 1000,
24             &commandVelocityReceived);
25           ros::spin();
26           delete pubPtr;
27         }
```

لیست (۶-۴) برنامه C++ به نام reverse_cmd_vel که دستور سرعت لاک پشت را معکوس می کند.

تنها دلیل، که این مثال مربوط به بخش نگا شت ا ست، این ا ست که شبیه ساز turtlesim به این پیام های معکوس شـده گوش نمی کند. در واقع، اجرای سـه نود مربوطه با دسـتور rosrun گراف شکل ۶,۲ را موجب می شود، از گراف مشخص ا ست که سیستم کار خواسته شده را انجام نمی دهد. چون د ستور سرعت م ستقیماً از teleop_turtle به turtlesim می رود، لاک پ شت در حالت معمول و معکوس نشده حرکت می کند.



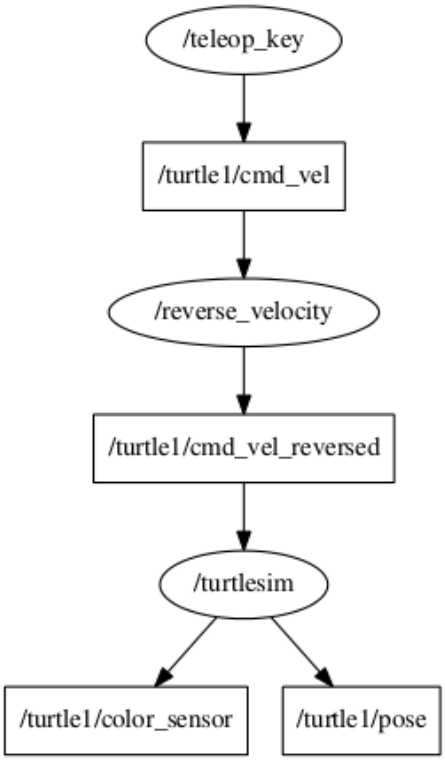
شکل (۶-۲) نتیجه ی گراف رآس از تلاشی نادرست برای استفاده از reverse_cmd_vel برای معکوس کردن لاک پشت turtlesim.

در این شرایط ، که نودی به تاپیک اشتباهی گوش می دهد، یک راه دقیق استفاده از نگاشت است. در این مورد، ما می توانیم نگاشتی را به turtlesim بفرستیم که turtle1/cmd_vel را با turtle1/cmd_vel_reversed عوض کند. لیست ۶,۵ فایل لانچی را نشان می دهد که هر سه نود را شروع می کند، شامل remap مناسب برای turtlesim_node. شکل ۶,۳ گراف درست از نتیجه را نشان می دهد.

```
1  <launch>
2   <node
3    pkg="turtlesim"
4    type="turtlesim_node"
5    name="turtlesim"
6   >
7    <remap
8     from="turtle1/cmd_vel"
9     to="turtle1/cmd_vel_reversed"
10          />
11        </node>
12        <node
13          pkg="turtlesim"
14          type="turtle_teleop_key"
15          name="teleop_key"
16          launch-prefix="xterm -e"
```

```
17          />
18          <node
19            pkg="agitr"
20            type="reverse_cmd_vel"
21            name="reverse_velocity"
22          />
23        </launch>
```

لیست (۶-۵) فایل لانچی با سه نود.



شکل (۶-۳) نتیجه گراف درست از reversed.launch. المان remap این امکان را فراهم می کند که نود به درستی ارتباط برقرار کند.

۶-۴- المان های دیگری از فایل لانچ

در این بخش ساختار های دیگری از roslaunch را معرفی می کنیم.[۱] برای روشن شدن این ویژگی ها به فایل لانچ لی ست ۶,۶ ارجاع خواهیم داد. این فایل لانچ ب سته به اینکه چگونه لانچ می شود، دو یا سه شبیه ساز turtlesim مستقل اجرا می کند.

```
<launch>
 <include
  file="$(find agitr)/doublesim.launch"
 />
 <arg
  name="use_sim3"
  default="0"
 />

        <group ns="sim3" if="$(arg use_sim3)" >
         <node
          name="turtlesim_node"
          pkg="turtlesim"
          type="turtlesim_node"
         />
         <node
          pkg="turtlesim"
          type="turtle_teleop_key"
          name="teleop_key"
          required="true"
          launch-prefix="xterm -e"
         />
        </group>
       </launch>
```

لیست (۶-۶) فایل لانچی به نام triplesim.launch که آرگومان های group و include و arg را مشخص می کند.

[۱] http://wiki.ros.org/ROS/Tutorials/Roslaunchtipsforlargerprojects

۶-۴-۱- اضافه کردن فایل های دیگر

برای اضـــافه کردن یک لانچ فایل دیگر، شـــامل همه ی نودها و پارامترهایش، از المان include استفاده کنید:[۱]

<include file="path-to-launch-file" />

این ویژگی فایل نیاز به مسیر کامل فایل اضافه شده دارد. چون وارد کردن این اطلاعات به صورت مستقیم سنگین و قابل تغییر است، بیشتر المان های include در عوض از find برای پیدا کردن پکیج استفاده می کنند، به جای اینکه صریحاً اسم پوشه را بیان کنند:

<include file="$(find package-name)/launch-file-name" />

آرگومان find به وسیله ی رشته ای از مسیر پکیج داده ذکر شده جایگزین می شود. اشاره به فایل لانچ مورد نظر از آن مســیر راحتتر اســت. برای مثال از این تکنیک برای اضــافه کردن مثال قبلی doublesim.launch استفاده می کنیم.

فراموش نکنید که roslaunch برای پیدا کردن فایل لانچ به وسـیله کامند لاین، زیرشـاخه های پکیج داده شده را نیز جستجو می کند. در صورتیکه، المان های include باید مسیر مشخص برای فایل مورد نظر را نام ببرند، و نمی توانند زیر شـــاخه ها را جســـتجو کنند. این تفاوت علت اخطار احتمالی ناشی از المان include بالا را بیان می کند، حتی اگر فراخوانی با roslaunch، با اسم پکیج و اسم فایل لانچ مشابه موفقیت آمیز باشد.

المان include همچنین ویژگی ns برای قراردادن محتویاتش درون یک فضـای نامی را پشـتیبانی می کند:

<include file=". . . " ns="namespace" />

این تنظیمات معمولاً انجام می شود، بخصوص زمانی که فایل لانچ درون پکیج دیگری اضافه می شود، و باید تقریباً مستقل از نودهای دیگر عمل کند.

[۱] http://wiki.ros.org/roslaunch/XML/include

۶-۴-۲- آرگومان های لانچ

برای کمک به قابل تنظیم کردن فایل لانچ، roslaunch آرگومان های لانچ را پشـــتیبانی می کند، که arguments و یا args خوانده می شـــوند. که تقریباً مشـــابه با متغییرهای محلی در برنامه های قابل اجرا عمل می کند.[۱] مزیتش این است که شما می توانید از تکرار کد جلوگیری کنید با نوشتن فایل های لانچ که از آرگومان هایی برای تعداد کمی از جزئیات که از یک اجرا به اجرای دیگر ممکن است تغییر کنند استفاده می کند. برای روشن شدن ایده، مثال فایل لانچ یک آرگومان، به نام use_sim3، استفاده می کند، تا مشخص کند سه تا یا دو تا کپی از turtlesim را اجرا کند.

گرچه اصـــلاحات آرگومان و پارامتر تاحدودی به صـــورت مشـــابه در خیلی از متون کامپیوتری استفاده شده اند، معنای آنها در رآس کاملاً متفاوت است. پارامترها مقادیری هستند که توسط سیستم فعال رآس استفاده می شوند، در سرور پارامترها ذخیره می شوند و توسط نودهای فعال به وســـیله ی تابع ros::param::get و توســـط کاربرها به وســـیله ی rosparam قابل دســـترس هســـتند (فصـــل هفتم.) در صـــورتیکه، آرگومان ها فقط زمانی درون فایل لانچ معنا دارند، مقادیرشان مستقیماً در دسترس نودها نیست.

مشخص کردن آرگومان ها: برای مشخص کردن آرگومان موجود می توانید از المان arg استفاده کنید:

```
<arg name="arg-name" />
```

اعلام کردن آرگومان بدین صورت الزامی نیست (تا زمانی که بخواهید مقدار یا پیش فرضی را بدان نسبت دهید- به قسمت بعدی مراجعه کنید)، اما ایده خوبی است چون برای خواننده ی انسانی مشخص می کند فایل لانچ انتظار چه آرگومان هایی را دارد.

نســـبت دادن مقادیر به آرگومان ها: باید به هر آرگومان اســـتفاده شـــده در فایل لانچ، مقداری نســـبت داده شـــود. چند راه برای این کار وجود دارد. شـــما می توانید مقداری در کامند لاین roslaunch ایجاد کنید:

```
roslaunch package-name launch-file-name arg-name:=arg-value
```

راه دیگر، شما می توانید مقادیر را به عنوان قسمتی از arg مشخص کنید، با استفاده یکی از این دو روش:

```
<arg name="arg-name" default="arg-value" />
<arg name="arg-name" value="arg-value" />
```

[۱] http://wiki.ros.org/roslaunch/XML/arg

تنها تفاوت بین این دو روش این اســـت که آرگومان کامند لاین می تواند مقدار پیش فرض (default) را تغییر دهد اما نمی تواند مقدار (value) را تغییر دهد. در این مثال، use_sim3 مقدار پیش فرض اش default صفر است، اما این مقدار پیش فرض می تواند توسط کامند لاین به صورت زیر تغییر کند:

roslaunch agitr triplesim.launch use_sim3:=1

اگر شـــما فایل لانچ را تغییر دهید، و default را با value تغییر دهید، کامند لاین خطا ایجاد می کند، چون مقادیر آرگومان ها مشخص شده با value نمی توانند تغییر کنند.

فر ستادن مقادیر آرگومان به فایل های لانچ ا ضافه شده: یک محدودیت تکنیک های گفته شده برای تنظیم آرگومان ها این است که هیچ ابزاری برای فرستادن آرگومان ها به فایل های لانچ وابـسته، که ما با المان include آنها را ا ضافه می کنیم، ندارند. این بـسیار مهم ا ست چون، بـسیار مشـــابه متغییرهای محلی (lexical)، آرگومان ها فقط برای فایل لانچ خودشـــان تعریف شـــده اند. آرگومان ها به فایل های لانچ اضافه شده منتقل نمی شوند.

راه حل ان است که المان arg به عنوان زیر شاخه ی المان include وارد شود، مانند:

```
<include file="path-to-launch-file">
   <arg name="arg-name" value="arg-value"/>
    ...
</include>
```

توجه داشـــته باشـــید این اســـتفاده از المان arg از مشـــخص کردن arg که قبلاً دیدیم فرق دارد. آرگومانهایی که بین دو تگ include آورده شـــده اند متعلق به فایل لانچ اضـــافه شـــده اند، نه برای لانچ فایلی که در آن ظاهر شـــده اند. چون هدف این اســـت که مقادیری برای آرگومان هایی مورد نیاز فایل اضافه شده منتشر کنیم، ویژگی value در اینجا به کار می آید.

یک حالت محتمل این است که هر دو فایل لانچ ـــ فایلی که اضافه شده و فایلی که اضافه کرده ـ چند آرگومان مـشترک دارند. در این حالت، ما ممکن ا ست این مقادیر را بدون تغییر نـسبت دهیم. المانی که اسم آرگومان مشابه را هر دو مکان استفاده می کند، مشابه زیر است:

```
<arg name="arg-name" value="$(arg arg-name)" />
```

در این مثال، اولینarg-name مانند همیـ شه، به آرگومان درون فایل لانچ ا ضافه شده برمی گردد. دومین arg-name به آرگومان خود فایل لانچ (اضافه کننده) بر می گردد. در نتیجه آرگومان در هر دو فایل لانچ مقدار یکسانی خواهند داشت.

۶-۴-۳- ساختن گروه ها

ویژگی نهایی فایل لانچ المان group است، که یک راه راحت برای سازمان دهی نودها در یک فایل لانچ بزرگ فراهم می کند.[۱] المان group می تواند دو هدف را انجام دهد:

- گروه ها می توانند نودهای مختلفی را در یک فضای نامی قرار دهند.

```
<group ns="namespace" />
...
</group>
```

هر نود درون گروه با فضای نامی داده شده اجرا می شود.

اگر یک نود گروه ویژگی ns خودش را داشته باشد، و آن نام (احتمالاً باید) یک اسم نسبی باشد، در نتیجه نود در فضای نامی تودرتو که فضای نامی خودش بعد از فضای نامی گروه می آید اجرا می شود. این قوانین، که از اسم های نسبی انتظار می رود، برای گروه های تودرتو هم اعمال می شود.

- گروه ها می توانند با توجه به شرایط نودها را فعال یا غیرفعال کنند.

```
<group if="0-or-1" />
...
</group>
```

اگر مقدار شرط if برابر یک باشد، در نتیجه المان های داخلی به صورت عادی اضافه می شوند. و اگر صفر باشد، المان های داخلی نادیده گرفته می شوند. ویژگی unless کاری مشابه، اما معکوس انجام می دهد.

```
<group unless="1-or-0" />
...
</group>
```

مطمئناً به ندرت صفر و یک مستقیماً در این ویژگی نوشته می شوند. با این وجود در ترکیب با ویژگی arg آنها ابزار قدرتمندی برای تنظیم کردن فایل های لانچ تشکیل داده اند.

توجه داشته باشید که صفر و یک تنها مقادیر قابل استفاده برای این ویژگی ها هستند. در عمل، عملگرهای بولین AND و OR که شما ممکن است انتظار داشته باشید کار نمی کنند.

این مثال یک گروه تک دارد که این دو هدف را ترکیب می کند. Group هر دو ویژگی ns (برای قرار دادن گروه دو نود در فضای نامی sim3) و ویژگی if (برای فعال و غیرفعال کردن سومین شبیه ساز براساس آرگومان use_sim3).

[۱] http://wiki.ros.org/roslaunch/XML/group

توجه داشته باشید group هیچ وقت واقعاً نیاز نیستند، همیشه راهی برای نوشتن ویژگی های if،unless ns ، به صـورت دسـتی برای هر المان، که ممکن اسـت بخواهیم اضـافه کنیم، وجود دارد. هرچند، گروه ها ممکن است تکرار را کاهش دهند و سازمان دهی فایل لانچ را به آسانی آشکارتر بکنند.

متـأسـفـانـه، فقط این سـه ویژگی می تواننـد توسـط group اعمـال شـونـد. برای مثال،output="screen" نمی تواند برای گروه تنظیم شود، و باید برای هر نود که می خواهیم به صورت مستقیم اعمال شود.

۶-۵- در ادامه

در این بخش، ما دیدیم چگونه نودها را تشـکیل دهیم که با فرسـتادن پیام ها با یکدیگر ارتباط برقرار کنند و چگونه نودهای زیادی را با تنظیمات پیچیده یک دفعه شـروع کنیم.فصـل بعد پارامترهای سـرور را معرفی می کند، که راه متمرکزی برای فراهم کردن اطلاعات تنظیمات نودها ایجاد می کند.

فـصل ۷ : پارامترها

در این فصل ما یاد می گیریم نودها را با استفاده ازپارامترها تنظیم کنیم.

علاوه بر پیام هایی که تا کنون مطالعه کردیم، رآس مکانیزم دیگری به نام پارمتر (parameter) برای دادن اطلاعات به نودها فراهم کرده اســـت. ایده این اســـت که یک ســـرور پارامتری مرکزی مجموعه ی همه ی مقادیر را دنبال کند (مقادیری مانند اعداد صحیح، اعداد اعشاری، رشته ها، یا داده ی های دیگر) که هر کدام با اسم رشته ای کوتاهی تعریف شده اند.[۲] [۱] چون پارامترها باید به صورت مداوم به وسیله ی نودهای علاقه مند به مقادیر شان مورد بازرسی قرار بگیرند، آنها بسیار برای اطلاعات تنظیم که زیاد در طول زمان تغییر نمی کنند مفید هستند.

در این فصل پارامترها تعریف می شوند، چگونه دسترسی به آنها را از کامند لاین، درون نودها و درون فایل های لانچ نشان می دهیم.

۷-۱- دسترسی به پارامترها از کامند لاین

بیاید با تعداد دستور شروع کنیم تا ببینیم پارامترها چگونه کار می کنند.

لیست پارامترها : برای دیدن لیست همه ی پارامترهای موجود، از دستور زیر استفاده کنید:[۳]

```
rosparam list
```

در کامپیوتر نویسنده، که نودی فعال نیست، خروجی بدین صورت است:

```
/rosdistro
/roslaunch/uris/host_donatello__38217
/rosversion
/run_id
```

هر کدام از این رشته ها یک نام است، مشخصاً یک نام منبع گراف جهانی (فصل ۵)، که سرور پارامتر به مقادیری نسبت داده است.

در ویرایش فعلی رآس، سرور پارامتر در واقع قسمتی از مستر است، در نتیجه به صورت خودکار با roscore یا roslaunch شروع می شود. تقریباً در همه ی موارد، سرور پارامتر در پشت صحنه کار می کند، و هیچ دلیلی برای فکر کردن در موردش وجود ندارد. هرچند در ذهن داشته باشید، که همه ی پارامترهای متعلق به سرور پارامترها هستند نه به هیچ نود خاصی. این بدین معنا است که پارامترها (حتی آنهایی که با اسم های خصوصی تشکیل شده اند) به حیاتشان ادامه می دهند حتی اگر نودی که آنها را تعیین کرده متوقف شود.

[۱] http://wiki.ros.org/roscpp/Overview/ParameterServer

[۲] http://wiki.ros.org/ParameterServer

[۳] http://wiki.ros.org/rosparam

سرکشی به پارامترها: برای درخوا ست مقدار یک پارامتر از سرور پارامتر، از د ستور rosparam get استفاده کنید:

rosparam get parameter_name

برای مثال، برای خواندن مقدار پارمتر rosdistro/ از دستور زیر استفاده کنید:

rosparam get /rosdistro

خروجی رشته ی indigo است، که تعجب آور هم نیست. این امکان هم وجود دارد که مقدار هر پارامتر درون یک فضای نامی بازیابی شود:

rosparam get namespace

برای مثال، با پرسیدن درمورد فضای نامی جهانی، ما می توانیم مقدار هر پارامتر را در لحظه ببینیم:

rosparam get /

در کامپیوتر نویسنده خروجی بدین صورت است:

rosdistro: indigo
roslaunch:
uris: host_donatello__38217: ’http://donatello:38217/’
rosversion: 1.11.9
run_id: e574a908-70c5-11e4-899e-60d819d10251

تنظیم پارامترها: برای نسبت دادن یک مقدار به یک پارامتر، از دستور زیر استفاده کنید:

rosparam set parameter_name parameter_value

این دستور می تواند مقادیر پارامترهای موجود را تغییر دهد و یا پارامترهای جدید بسازد. برای مثال، این د ستورات پارامترهای ر شته ای ایجاد می کند که تنظیمات کمد لباس یک گروه خا صی از اردک های کارتونی را ذخیره می کند.

Rosparam set /duck_colors/huey red
Rosparam set /duck_colors/dewey blue
Rosparam set /duck_colors/louie green
Rosparam set /duck_colors/webby pink

از طرف دیگر، ما می توانیم پارامترهای مختلفی را در یک فضای نامی یکجا تنظیم کنیم:

rosparam set namespace values

مقادیر باید در یک دیکشنری YAML مشخص شده باشند که اسم های پارامترها را به مقادیر نگاشت کند. اینجا یک مثال که تأثیری مشابه چهار دستور بالا دارد می بینیم:

rosparam set /duck_colors "huey: red
dewey: blue
louie: green
webby: pink"

توجه داشته باشید که این دستور به کاراکتر خط جدید درون دستور خودش نیاز دارد. این یک م شکل نی ست چون علامت نقل قول (") به bash علامت می دهد که د ستور هنوز تمام ن شده

ا ست. با فشار دادن Enter وقتی که علامت نقل قول باز ا ست یک کاراکتر خط جدید وارد می شود به جای اینکه دستور اجرا شود.

فضـای خالی بعد از علامت دو نطقه خیلی مهم اسـت، برای اینکه مطمئن شـویم rosparam این قسـمت را به عنوان مجموعه ای از پارامترها در فضـای نامی duck_colors/ در نظر می گیرد، به جای اینکه آن را یک پارامتر رشـته ای به نام duck_colors در فضـای نامی جهانی در نظر بگیرد.

سـاختن و پرکردن فایل های پارامتر: برای ذخیره کردن پارمترها در فضـای نامی، به فرمت YAML، درون یک فایل، از rosparam dump استفاده کنید:

rosparam dump filename namespace

مخالف dump عبارت load ا ست، که پارامترها را از یک فایل می خواند و آنها را به سرور پارمترها اضافه می کند:

rosparam load filename namespace

برای هر کدام از این دستورات، آرگومان های فضای نامی دلبخواه هستند، و به صورت پیش فرض فضـای نامی جهانی (/) اسـت. ترکیب dump و load می تواند برای امتحان مفید باشـد، چون این ترکیب راه سـریعی برای دیدن یکجای پارامترها در یک زمان مشـخص، و برای ایجاد دوباره همین حالت فراهم می کند.

۷-۲- مثال: پارامترها در turtlesim

برای یک مثال عینی تر از اینکه چگونه پارامترها می توانند مفید باشـند، بیاید ببینیم چگونه turtlesim آنها را استفاده می کند. اگر roscore و turtlesim_node را شروع کنید، و را درخواست rosparam list کنید، خروجی مانند زیر می بینید:

```
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_donatello__59636
/rosversion
/run_id
```

ما تا اینجا چهار پارامتر آخر را دیده ایم، که به و سیله ی م ستر ایجاد می شوند. بعلاوه، اینگونه به نظر می رسد که turtlesim_node سه پارامتر ایجاد کرده است. نامهایشان نشان می دهد که turtlesim از چه رنگی برای پشت زمینه استفاده می کند، رنگ را در سه کانال جداگانه قرمز، سبز، و آبی نشان می دهد.

این نشان می دهد که نودها می توانند مقادیر پارامتر را ایجاد یا تغییر دهند. در این مورد، turtlesim_node این سه پارامتر را در م قداردهی اول یه تنظیم می ک ند. در این راستا، turtlesim_node غیر معمول است، چرا که مقدار دهی اولیه آن هیچ مقدار که ممکن است برای آن پارامترها م شخص شده با شد نمی تواند تغییر دهد. این ا ست که، هر turtlesim_node با یک پس زمینه آبی رنگ شروع می شود، حداقل برای یک زمان کوتاه، بدون در نظر گرفتن هر قدمی که ممکن است برای مشخص کردن یک رنگ شروع متفاوت برداشته باشیم.

- یک روش بهتر برای turtlesim، و یک مثال بهتر برای اینکه چگونه نودهای واقعی رآس چگونه کار می کنند - این است که ابتدا چک کنند ببینید آن پارامترها وجود دارند، و فقط درصورت عدم وجود پارامترها مقدار پیش فرض رنگ آبی را بدان نسبت دهد.

خواندن رنگ پشت زمینه: ما می توانیم مقادیر پارمترهای پشت زمینه را با استفاده از rosparam get بدست بیاوریم:

```
rosparam get /background_r
rosparam get /background_g
rosparam get /background_b
```

مقادیر برگشتی از دستور ۶۹، ۸۶، ۲۵۵ هستند. چون مقادیر اعداد صحیح نسبتاً کوچکی هستند، یک حدس خوب (که درست هم هست) این است که هر کانال یک عدد صحیح ۸ بیتی است، بین ۰ و ۲۵۵. بنابراین، پشت زمینه ی به صورت پیش فرض(۲۵۵،۸۶،۶۹) است، مربوط به رنگ آبی تیره که ما استفاده می کنیم.

تنظیم رنگ پشت زمینه: فرض کنیم می خواهیم این رنگ پشت زمینه را از آبی به زرد رو شن تغییر دهیم. ما شاید سعی کنیم با تغییر مقادیر پارامتر بعد از شروع نود turtlesim این کار را انجام دهیم:

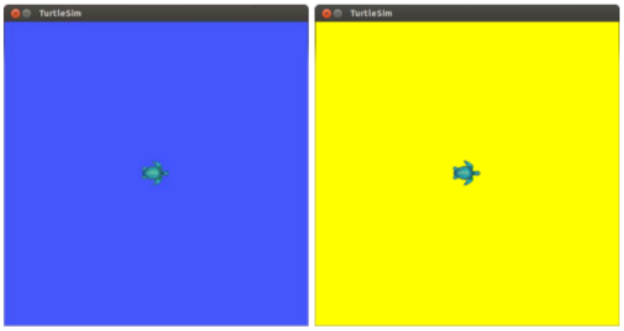
```
rosparam set /background_r 255
rosparam set /background_g 255
rosparam set /background_b 0
```

هرچند، حتی بعد از تنظیم این پارامترها، پشت زمینه همان رنگ باقی می ماند. چرا؟ پاسخ این است که turtlesim_node مقادیر این پارامترها را فقط وقتی می خواند که سرویس /clear فراخوانده شده باشد. یک راه برای فراخواندن سرویس /clear دستور زیر است:

rosservice call /clear

بعد از تکمیل فراخوانی سرویس، رنگ پشت زمینه بلاخره به صورت مناسب تغییر می کند. شکل ۷,۱ این اثر را نشان می دهد.

نکته ی قابل توجه در اینجا این است که مقادیر به روز شده ی پارامترهای به صورت خودکار به نودها داده نمی شوند. در صورتیکه، نودهایی که به تغییر بعضی یا همه ی پارامترها اهمیت می دهند باید به صورت واضح از سرور پارامتر درخواست آن مقادیر را بکنند. همان طور که، اگر ما انتظار داشته باشیم که مقادیر پارامترهای مورد استفاده ی هر نود را تغییر دهیم، ما باید مواظب باشیم چگونه (و یا اگر) این نود پارامترهایش را دوباره نمایش می دهد. (بیشتر اوقات، اما نه برای turtlesim، جواب برا ساس زیر سیستمی به نام dynamic_reconfigure ا ست، که ما اینجا آن را پوشش نمی دهیم. [۱])



شکل (۷-۱) قبل (چپ) و بعد (راست) تغییر رنگ پشت زمینه ی نود turtlesim.

این امکان برای هر نود وجود دراد که از سرور پارامترها بخواهد هر وفت پارامتری تغییر کرد مقدارش را برایش بفرستد، با استفاده از ros::param::getCached به جای ros::param::get. هرچند، این روش فقط می خواهد کارآیی را بهبود ببخشد، و نیاز نود برای چک کردن مقدار پارامتر را حذف نمی کند.

۷-۳- دسترسی به پارامتر از C++:

ارتباط C++ با پارامترهای رآس کاملاً مستقیم است:[۲]

```
void ros::param::set(parameter_name, input_value);
bool ros::param::get(parameter_name, output_value);
```

[۱] http://wiki.ros.org/dynamic_reconfigure

[۲] http://wiki.ros.org/roscpp_tutorials/Tutorials/Parameters

در هر دو مورد، نام پارامتر یک رشته است، که می تواند جهانی، نسبی، یا خصوصی باشد. مقدار ورودی برای set می تواند int، bool،std::string و یا double باشـد؛ مقدار خروجی برای get باید یک متغییر (که از مرجع فرسـتاده می شـود) از این نوع ها باشـد. تابع get مقدار true را بر می گرداند اگر مقدار با موفقیت خوانده شده باشد و false را بر می گرداند اگر یک اشکالی وجود داشته باشد، معمولاً نشان می دهد که به پارامتر خواسته شده مقداری نسبت داده نشده است. برای دیدن این توابع در عمل، بیاید به دو مثال زیر نگاهی بیاندازیم.

- لیست ۷,۱ ros::param::set را روشن می کند. عدد صحیحی را به هر سه پارامتر رنگ پشت زمینه ی turtlesim نسبت می دهد. این برنامه شامل کدی است که مطمئن شود که نود turtle-sim بعد از انتظار برای سرویس/clear شروع شده است (که برای اطمینان از اینکه turtlesim مقادیر که ما اینجا تنظیم کردیم را بازنویسـی نمی کند لازم اسـت.) واین سرویس را فرا خواند که turtlesim را مجبور کند مقدار جدید که ما تنظیم کردیم را بخواند. (تمرکز ما اینجا روی خود پارامترهاست، فصل ۸ را برای سرویس ها ببینید.)

```
1  // This program waits for a turtlesim to start up, and
2  // changes its background color.
3  #include <ros/ros.h>
4  #include <std_srvs/Empty.h>
5
6  int main(int argc, char **argv) {
7    ros::init(argc, argv, "set_bg_color");
8    ros::NodeHandle nh;
9
10           // Wait until the clear service is available, which
11           // indicates that turtlesim has started up, and has
12           // set the background color parameters.
13           ros::service::waitForService("clear");
14
15           // Set the background color for turtlesim,
16           // overriding the default blue color.
17           ros::param::set("background_r", 255);
18           ros::param::set("background_g", 255);
19           ros::param::set("background_b", 0);
20
21           // Get turtlesim to pick up the new parameter values.
22           ros::ServiceClient clearClient
```

```
23              = nh.serviceClient<std_srvs::Empty>("/clear");
24            std_srvs::Empty srv;
25            clearClient.call(srv);
26
27          }
```

لیست (۷-۱) یک برنامه C++ به نام set_bg_color.cpp که رنگ پشت زمینه ی پنجره ی turtlesim را تنظیم می کند.

- لیست ۷,۲ مثالی از ros::param::get را نشان می دهد. این برنامه مثال pubvel (لیست ۳,۴) را بسط می دهد. پارامتر خصوصی اشاری به نام max_vel را می خواند و از این مقدار برای بزرگتر یا کوچکتر کردن سرعت های خطی که به صورت تصادفی تشکیل شده اند استفاده می کند.
- این برنامه به مقداری برای پارامتر max_vel در فضای نامی خصوصی اش نیاز دارد، که باید قبل از شروع برنامه تنظیم شود:

rosparam set /publish_velocity/max_vel 0.1

اگر آن پارامتر در دسترس نباشد، برنامه یک خطای مهلک (fatal) ایجاد می کند و متوقف می شود.

```
1   // This program publishes random velocity commands, using
2   // a maximum linear velocity read from a parameter.
3   #include <ros/ros.h>
4   #include <geometry_msgs/Twist.h>
5   #include <stdlib.h>
6   int main(int argc, char **argv) {
7     ros::init(argc, argv, "publish_velocity");
8     ros::NodeHandle nh;
9     ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
10                "turtle1/cmd_vel", 1000);
11            srand(time(0));
12            // Get the maximum velocity parameter.
13            const std::string PARAM_NAME = "~max_vel";
14            double maxVel;
15            bool ok = ros::param::get(PARAM_NAME, maxVel);
16            if(!ok) {
17              ROS_FATAL_STREAM("Could not get parameter "
18                << PARAM_NAME);
19              exit(1);
20            }
21            ros::Rate rate(2);
22            while(ros::ok()) {
23              // Create and send a random velocity command.
24              geometry_msgs::Twist msg;
25              msg.linear.x = maxVel*double(rand())/double(RAND_MAX);
26              msg.angular.z = 2*double(rand())/double(RAND_MAX)-1;
27              pub.publish(msg);
28              // Wait until it's time for another iteration.
29              rate.sleep();
30            }
31          }
```

لیست (۷-۲) برنامه C++ به نام pubvel_with_max.cpp که برنامه ی اصلی pubvel.cpp را، برای خواندن ماکزیمم سرعت خطی اش از یک پارامتر، بسط داده است.

۷-۴- تنظیم پارامترها در فایل های لانچ

این کار از لحاظ فنی ممکن است (اما مقداری بی نظم است) که با استفاده از کامند لاین که از حالتی شبیه نگاشت استفاده می کند، یک پارامتر خصوصی را به یک نود نسبت دهیم با اضافه کردن خط تیره (_) در اول اسم:

_param-name:=param-value

این نوع آرگومان به فراخوانی از ros::param::set به وسیله ی ros::init تبدیل می شوند، با عوض کردن _ با ~ که یک شکل درستی از پارامتر خصوصی را شکل دهد. برای مثال، ما می توانیم pubvel_with_max را با استفاده از این دستور شروع کنیم:

rosrun agitr pubvel_with_max _max_vel:=1

راه متداول دیگر برای تنظیم پارامترها استفاده از فایل لانچ است.

تنظیم پارامترها : برای درخواست تنظیم مقدار پارامتر از roslaunch، از المان param استفاده کنید:[۱]

```
<param name="param-name" value="param-value" />
```

این المان، همان طور که انتظار می رود، مقدار مشخص شده را به پارامتر با اسم مشخص شده نسبت می دهد. اسم پارامتر باید، مثل همیشه، یک اسم نسبی باشد. برای مثال، این تکیه از فایل لانچ معادل دستور rosparam set صفحه ی ۱۰۷ است:

```
<group ns="duck_colors">
<param name="huey" value="red" />
<param name="dewey" value="blue" />
<param name="louie" value="green" />
<param name="webby" value="pink" />
</group>
```

تنظیم پارامترهای خصوصی: یک گزینه ی دیگر اضافه کردن المان param به عنوان زیر شاخه ی المان نود است:

```
<node . . . >
<param name="param-name" value="param-value" />
...
</node>
```

با این ساختار، اسم پارامترها به عنوان اسم خصوصی برای آن نود در نظر گرفته می شوند.

[۱] http://wiki.ros.org/roslaunch/XML/param

این مورد یک استثناء برای قوانین متداول اسم گذاری است. اسم پارامترهای مشخص شده در المان های param که زیر شاخه ای از المان node هستند همیشه به عنوان اسم خصوصی در نظر گرفته می شوند، بدون توجه به اینکه ابتدای آنها با ~ یا / شروع می شود.

برای مثال، شاید ما از کدی مشابه زیر استفاده کنیم تا نود pubvel_with_max را شروع کنیم که پارامتر خصوصی max_vel به درستی در آن تنظیم شده است:

```
<node
pkg="agitr"
type="pubvel_with_max"
name="publish_velocity"
/>
<param name="max_vel" value="3" />
</node>
```

لیست ۷,۳ یک فایل لانچ کامل را نشان می دهد که turtlesim و دو مثال ما را شروع می کند. نتیجه باید لاک پشتی سریع در پشت زمینه ای به رنگ زرد را نشان دهد.

خواندن پارامترها از یک فایل: و در آخر، فایل های لانچ دستوری معادل rosparam load را نیز پشتیبانی می کند، تا پارامترهای زیادی را یک دفعه تنظیم کند.[۱]

```
<rosparam command="load" file="path-to-param-file" />
```

لیست فایل پارمتر در اینجا معمولاً به وسیله ی rosparam dump تشکیل می شود. مانند مراجع دیگر برای مشخص کردن فایل ها (مانند المان include در بخش ۶,۵,۱) این معمول است که از find برای مشخص کردن اسم فایل نسبت به پوشه ی یک پکیج استفاده کنیم.

```
<rosparam
command="load"
file="$(find package-name)/param-file"
/>
```

```
1  <launch>
2   <node
3    pkg="turtlesim"
4    type="turtlesim_node"
5    name="turtlesim"
6   />
7   <node
8    pkg="agitr"
9    type="pubvel_with_max"
10          name="publish_velocity"
```

[۱] http://wiki.ros.org/roslaunch/XML/rosparam

```
11          >
12            <param name="max_vel" value="3" />
13          </node>
14          <node
15            pkg="agitr"
16            type="set_bg_color"
17            name="set_bg_color"
18          />
19        </launch>
```

لیست (۷-۳) یک فایل لانچ به نام fast_yellow.launch که برنامه های لیست ۷,۱ و ۷,۲ را شروع می کند و پارامتر max_vel تنظیم می کند.

در کنار rosparam load، این امکانات برای تست کردن مفید هستند، چون به ما اجازه می دهد پارامترهایی که در زمان مشخصی در گذشته تغییر کرده اند را دوباره بسازیم.

۷-۵- در ادامه

پارامترها تقریباً ایده ی ساده ای هستند که می تواند انعطاف قابل توجه و قابلیت تنظیم بالایی برای نودهای رآس ایجاد کند. فصل بعدی با آخرین مکانیزیم ارتباط به نام سرویس آشنا می شویم، که جریان اطلاعات به صورت تک به تک و دوطرفه را پیاده سازی می کند.

فـصل ۸ : سرویس ها

در این فصل ما یاد می گیریم سرویس ها را فرا بخوانیم و به سرویس ها پاسخ بدهیم.

سرویس های و پاسخ به درخواست سرویس ها

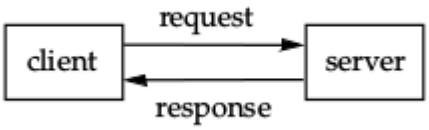
در فصل های ۲ و ۳ ما نشان دادیم که چگونه پیام ها بین نودها جابه جا می شوند. گرچه استفاده از پیام ها یک روش ارتباط در رآس هســـت، پیام ها محدودیت هایی دارند. در این فصـــل روش دیگری برای ارتباط که فراخوانی ســـرویس service call نام دارد را معرفی می کنیم. فراخوانی سرویس با پیام دو تفاوت دارد:

- فراخوانی سرویس دو طرفه است. یک نود یک پیام به نود دیگر می فرستد و منتظر پاسخ می ماند. اطلاعات دو طرفه جریان دارد. برخلاف پیام که وقتی منتشر می شود، پاسخی وجود ندارد و حتی هیچ ضمانتی وجود ندارد که نودی به پیام گوش می دهد.
- فراخوانی سرویس برای یک ارتباط یک به یک است. هر فراخوانی سرویس به وسیله ی یک نود تنظیم و مقدار دهی اولیه می شود، و پاسخ به همان نود بر می گردد. در صورتیکه، هر پیام مربوط به یک پیام است که ممکن است منتشر کننده های زیادی و شنونده های subscriber زیادی داشته باشد.

در کنار همه ی این تفاوت ها (خیلی مهم)، ســـرویس و پیام مشـــابه هم هســـتند. ما در این بخش نشان می دهیم چگونه یک سرویس را به وسیله command line فراخوانی یا سرکشی کنیم، و چگونه نودی بنویسیم که به عنوان مشتری سرویس یا سرور کار کند.

۸-۱- واژه شناسی سرویس ها

در اینجا مفاهیم اولیه فراخوانی سرویس را تعریف می کنیم:



ایده این است که نود مشتری- کلاینت (client) یک درخواست به نود سرور می فرستد و منتظر پاسخ می شود. سرور، که درخواست را دریافت کرد، چند کار انجام میدهد (محاسباتی انجام می دهد، سخت افزار یا نرم افزار را تنظیم می کند، رفتارش را تغییر می دهد و غیره) وداده هایی را به عنوان پاسخ به کلاینت بر می گرداند.

نوع محتوی درخوا ست و پا سخ با نوع داده ی سرویس (service data type) م شخص می شود، مشابه نوع پیام که مشخص می کردیم. مانند یک نوع پیام، نوع داده سرویس با نام فیلدها مشخص می شود. تنها تفاوت اینجا ست که نوع داده ی سرویس دو بخش دارد، درخوا ست (که کلاینت به سرور می فرستد) و پاسخ (که از سرور به کلاینت باز گرداننده می شود.)

۸-۲- پیدا کردن و فراخوانی سرویس با command line

گرچه ســـرویس ها معمولا با کد در نود تعریف می شـــوند اما command line های کمی نیز برای ارتباط با آنها وجود دارد. استفاده از این کامند لاین ها کمک می کند که عملکرد فراخوانی سرویس ها را راحتتر درک کنیم.

گوش کردن به همه سرویس ها: شما می توانید یک لیستی از سرویس ها که در حال حاضر فعال هستند را با دستور زیر ببینید:[۱]

rosservice list

در کامپیوتر نویسنده که فقط نود turtlesim اجرا شده است، لیست سرویس ها به صورت زیر است:

/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level

هر خط نام یک سرویس را نشان می دهد که می تواند فراخوانی شود. نام سرویس ها هم نام تابع گرافی هســـتند. و مانند همه ی نام منابع گرافی می تواند به صـورت ســرآســری (global)، وابســته (relative)، خصو صی (private) م شخص شود. خروجی رآس سرویس لیست نام کامل سرا سری نشان داده می شود.

سرویس های این مثال ، و به صورت کلی ب سیاری از سرویس های رآس، به دو د سته ا سا سی تقسیم می شوند.

[۱] http://wiki.ros.org/rosservice

- دســـته ی اول، مانند get_loggers و set_logger_level در لیســـت بالا، برای گرفتن اطلاعات و فرســتادن اطلاعات به نود مشــخص اســتفاده می شــوند. این نوع ســرویس ها معمولا نام نود خود شان را به عنوان پیشوند (namespace) استفاده می کنند تا از تداخل نام ها جلوگیری کنند و به نودها اجازه بدهند که ســـرویس های خودشـــان را با نام های داخلی (private) مانند get_loggers یا set_logger_level تعریف کنند. (بخش ۴,۵ را برای جزییات بیشتر ببینید.)
- دســـته دوم، ویژگی های عمومی تری دارند که به یک نود مشـــخص بر نمی گردد. برای مثال ســرویس spawn، که یک لاک پشــت شــبیه ســازی جدید درســت می کند، با نود turtlesim استفاده شده است. گرچه در سیستم های دیگر این سرویس می تواند به وسیله ی نود دیگری استفاده شود. وقتی ما spawn را فراخوانی می کنیم ما فقط اهمیت می دهیم که یک لاک پشــت جدید بســازیم و اهمیتی نمی دهیم که چه نودی با آن کار می کند. همه ی سرویس های لیست بالا به جزget_loggers وset_logger_level در این دسته هستند. این نوع سرویس ها معمولا اسمی دارند که کاربردشان را مشخص می کند، اما به اسم نود مشخصی اشاره نکند.

دیدن لیست سرویس ها با نود : برای دیدن سرویسهای یک نود مشخص، از رآس اینفو استفاده می کنیم:

rosnode info node-name

برای مثال لیست سرویس های turtlesim:

* /turtle1/teleport_absolute
* /turtlesim/get_loggers
* /turtlesim/set_logger_level
* /reset
* /spawn
* /clear
* /turtle1/set_pen
* /turtle1/teleport_relative
* /kill

این نشان می دهد بیشتر سرویس هایی که اکنون فعال هستند توسط نود turtlesim در دسترس قرار گرفته است. (تنها دو سرویس لاگ با rosout استفاده می شود.)

پیدا کردن نودی که یک ســـرویس را ارائه می دهد: برای اجرای درخواســـت برعکس(که چه نودی یک سرویس مشخص را ایجاد می کند) از دستور زیر استفاده کنید:

rosservice node service-name

همان طور که انتظار می رود، خروجی دستور بالا برای سرویس های توی لیست turtlesim/ یا rosout/ خواهد بود.

پیدا کردن نوع داده ای یک سرویس: شما نوع داده ی یک سرویس را با دستور زیر می توانید بدست آورید:

rosservice info service-name

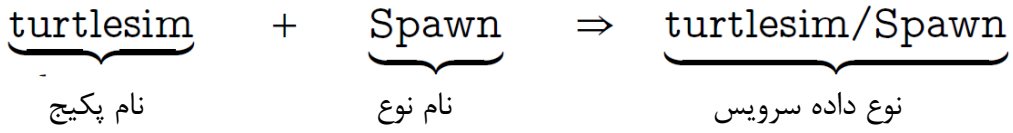
برای مثال:

rosservice info /spawn

خروجی:

Node: /turtlesim
URI: rosrpc://donatello:47441
Type: turtlesim/Spawn
Args: x y theta name

نوع داده سرویس spawn/ همان طور که می بنید turtlesim/Spawn است. مانند نوع پیام، نوع داده یک سرویس دو قسمت دارد، یک نام پکیج که نوع داده به آن تعلق دارد و نام خود نوع داده.



نوع داده سرویس همیشه به همین صورت کامل رفرنس داده می شود.

بازرسی انواع داده سرویس: برخی جزئیات در مورد نوع داده های سرویس را می توان با دستور rossrv به دست آورد:

rossrv show service-data-type-name

برای مثال

rossrv show turtlesim/Spawn

خروجی این دستور:

float32 x
float32 y
float32 theta
string name
---
string name

در این بخش، داده های قبل خط تیره (---) عناصر درخواست هستند. این اطلاعاتی است که نود کلاینت (درخواست کننده) به نود سرور می فرستد. هرچیزی بعد از خط تیره (---) پاسخ است، یا به عبارتی اطلاعاتی است که سرور، وقتی کارش تمام شد، به درخواست کننده برمی گرداند.

مراقب تفاوت rosservice و rossrv باشید. اولی برای ارتباط با سرویس ها است که به وسیله ی تعدادی از نودها فراهم شده اند. دومی برای درخواست نوع داده ی سرویس استفاده می شود، چه در حال حاضر سرویسی از آن نوع فعال باشد یا نه. این تفاوت مانند تفاوت بین rostopic و rosmsg است:

سرویس ها	تاپیک ها	
rosservice	rostopic	**موارد فعال**
rossrv	rosmsg	**نوع داده ها**

توجه داشته باشید که درخواست و پاسخ هر دو می توانند خالی باشند. برای مثال سرویس /reset که توسط نود turtlesim ایجاد می شود و از نوع std_srvs/Empty است، قسمت درخواست و پاسخ اش هر دو خالی است. تقریبا شبیه تابع در C++ که می توان بدون ورودی و بدون خروجی (return void) تعریف شود. در واقع هیچ اطلاعاتی وارد یا خارج نمی شود، اما موارد مفیدی (که می توانند تأثیر جانبی داشته باشد) اجرا شود.

فراخوانی سرویس ها از command line : برای اینکه حس کنید که سرویس ها چگونه کار می کنند، می توانید با استفاده از command line آنها را فراخوانی کنید.

rosservice call <اسم سرویس> <محتوی درخواست>

در قسمت محتوی درخواست باید مقادیر برای هر فیلد هر درخواست، همان طور که rossrv به شما نمایش می دهد، لیست شود. برای مثال:

rosservice call /spawn 3 3 0 Mikey

در اثر فراخوانی این سرویس یک لاک پشت به نام میکی در مختصات (x,y) = (۳،۳)، در جهت θ = ۰ در شبیه سازی که فعال است ساخته می شود.

این لاک پشت جدید با منابع خودش شامل pose , cmd_vel و تاپیک color_sensor و سرویس های teleport_relative , teleport_absolute , set_pen ایجاد می شود. این منابع فعال با پیشوند namespace قابل فراخوانی هستند مثلا در این مثال با پیشوند Mikey. این منابع اضافه بر منابع مفیدی هستند که در namespace لاک پشت۱ turtle1 موجود است. و در نتیجه نودهای دیگر می توانند هر لاک پشت را به صورت جداگانه کنترل کنند. این مورد به خوبی نشان می دهد که چگونه namespace ها می توانند از تداخل ها جلوگیری کنند.

خروجی فراخوانی رآس سرویس پاسخ سرور را نشان می دهد. برای مثال پاسخ باید اینگونه باشد:

name: Mikey

در این مورد، سرور نام لاک پشت جدید را به عنوان پاسخ برمی گرداند.

بعلاوه سرور به درخواست کننده می گویید که فراخوانی موفق بوده یا نه. برای مثال در turtlesim، هر لاک پشت باید یک اسم تک داشته باشد. اگر ما فراخوانی سرویس بالا را دوبار انجام دهیم، اولین مورد باید موفق باشد ولی دومی باید خطای زیر را نشان دهد:

ERROR: service [/spawn] responded with an error:

این خطا به خاطر این رخ می دهد که ما سعی کردیم دو لاک پشت با یک اسم درست کنیم.

این خطا با : خاتمه یافته است چون turtlesim پیام خطای خالی برگردانده است. زیر ساخت ها امکان برگردان یک پیام خطای کوچک در صورت موفق نبودن فراخوانی سرویس را فراهم می کند، اما کتابخانه ی C++ درخواست کننده، که نود turtlesim استفاده می کند، راه ساده ای برای برگرداندن پیام خطای خالی را فراهم نمی کند.

۸-۳- یک برنامه درخواست کننده

فراخوانی سرویس ها از command line ها ابزاری دمِ دستی برای فهمیدن و یا برای تست های موقتی است، اما به وضوح فراخوانی سرویس ها از کدها بسیار مفید است.[۱] لیست ۸,۱ یک مثال نشان می دهد. این مثال همه ی عناصر اولیه برای یک برنامه درخواست کننده ی سرویس را نشان می دهد.

مشخص کردن نوع درخواست و پاسخ : مانند نوع پیام (بخش ۳,۳,۱) هر داده سرویس یک سربرگ مربوطه در C++ دارد که باید اضافه شود.

#include <package_name/type_name.h>

در این مثال ما می گوییم:

#include <turtlesim/Spawn.h>

تا یک کلاس به نام turtlesim::Spawn از سرویسی که می خواهیم فراخوانی کنیم، اضافه کنیم. این کلاس نوع داده، شامل هر دو قسمت درخواست و پاسخ (request and response)، را تعریف می کند.

ساختن یک شی درخواست : بعد از تنظیمات اولیه ی خود نود (ros::init و ساختن NodeHandle)، برنامه ی ما باید یک شی از نوع ros::ServiceClient بسازد، که وظیفه اش فراخوانی سرویس است. تعریف ros::ServiceClient به صورت زیر است:

[۱] http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(C++)

ros::ServiceClient client = نود هندلر.serviceClient<نوع سرویس>(نام سرویس)

این خط سه قسمت مهم دارد.

- نود هندلر همان شـــیء معمول ros::NodeHandle اســـت. ما متد service-Client آن را فراخوانی می کنیم.
- نوع سرویس نام شیء سرویسی است که در سربرگ در بالا تعریف شده است. در این مثال منظور turtlesim::Spawn است.
- نام سرویس یک اسم دلخواه از نوع string است که می خواهیم سرویس را به آن نام صدا بزنیم. این اسم باید یک اسم نسبی باشد، اما می توان از یک اسم سراسری هم استفاده کرد. در مثال یک اسم مربوط یعنی "spawn" استفاده شده است.

به طور معمول ســاختن این شــیء نســبتا کم هزینه اســت چون کار خاصــی به جز ذخیره کردن جزئیاتی در مورد سرویسی که ما می خواهیم فراخوانی کنیم انجام نمی دهد.

```
1  // This program spawns a new turtlesim turtle by calling
2  // the appropriate service.
3  #include <ros/ros.h>
4  // The srv class for the service.
5  #include <turtlesim/Spawn.h>
6  int main(int argc, char **argv) {
7    ros::init(argc, argv, "spawn_turtle");
8    ros::NodeHandle nh;
9    // Create a client object for the spawn service.  This
10           // needs to know the data type of the service and its
11           // name.
12           ros::ServiceClient spawnClient
13             = nh.serviceClient<turtlesim::Spawn>("spawn");
14           // Create the request and response objects.
15           turtlesim::Spawn::Request req;
16           turtlesim::Spawn::Response resp;
17           // Fill in the request data members.
18           req.x = 2;
19           req.y = 3;
```

```
20          req.theta = M_PI / 2;
21          req.name = "Leo";
22          // Actually call the service.  This won't return until
23          // the service is complete.
24          bool success = spawnClient.call(req, resp);
25          // Check for success and use the response.
26          if(success) {
27            ROS_INFO_STREAM("Spawned a turtle named "
28              << resp.name);
29          } else {
30            ROS_ERROR_STREAM("Failed to spawn.");
31          }
32        }
```

لیست (۸-۱) برنامه spawn_turtle.cpp که یک سرویس را فرا می خواند.

توجه داشته باشید که برای ساختن ros::ServiceClient به سایز ذخیره کننده نیاز ندارید، برخلاف ros::Publisher. این تفاوت به این دلیل ا ست که تا زمانی که پا سخی نر سد فراخوانی سرویس چیزی را برنمی گرداند. چون درخواست کننده منتظر می ماند تا فراخوانی سرویس کامل شود. در نتیجه نیازی به ذخیره کننده نداریم که فراخوانی های بعدی را در خود نگه دارد.

به وجود آوردن شی های درخوا ست و پا سخ : وقتی ros::ServiceClient ساخته شد، قدم بعدی ساختن یک شیء درخواست(request) است که شامل داده هایی باشد که به سرور فرستاده می شود. در سربرگ کلاس هایی جداگانه ای از پا سخ و درخوا ست نوع داده سرویس ا ضافه می کنیم که به ترتیب Response و Request خوانده می شوند. این کلاس ها باید براساس نام پکیج و نوع سرویس مانند زیر رفرنس داده شوند:

package_name::service_type::Request
package_name::service_type::Response

هر کدام از کلاس ها عضوهایی متناسب با فیلد نوع سرویس دارند. (یادآوری که rossrv show می تواند لیست آن فیلدها را و نوع داده هایشان را نشان دهد.) این فیلدها به نوع داده های C++ به مانند فیلد پیام ها میتواند تصویر شوند. (تبدیل شوند) ساختار درخواست مقادیر به طور پیش فرض بدون مفهوم برای این فیلدها، چون ما باید یک مقداری را به هر فیلدی نسبت بدهیم. در این مثال،

ما یک شـیء turtlesim::Spawn::Request می سـازیم و مقادیری به x, y, theta و اسـم فیلدها نسبت می دهیم.
ما به یک شـیء Response نیاز داریم در این مثال، یک turtlesim::Spawn::Response اما چون این اطلاعات باید از سرور بیاید، ما نباید عضوهای آنها را مقدار دهی کنیم.

فایلهای سربرگ نوع سرویس همچنین یک کلاس (یک ساختار) با نام زیر را تعریف می کند.
package_name::service_type
که شامل درخواست و پاسخ به عنوان اعضاء است. یک شیء از این کلاس معمولا یک srv نامیده می شود. اگر شما ترجیح می دهید (مانند بسیاری از نویسنده های سایت های آموزشی آنلاین) می توانید یک متد call که در زیر تعریف شـــده به این کلاس برگردانید، به جای تعریف کردن شیء Request و Response جداگانه.

فراخواندن سرویس: وقتی ما یک ServiceClient ،یک درخوا ست کامل و یک پا سخ داریم، می توانیم یک سرویس را فراخوانیم:

```
bool success = service_client.call(request, response);
```

این روش کار ا صلی پیدا کردن نود سرور، فر ستادن داده ی درخوا ست، انتظار برای پا سخ و ذخیره ی داده ی پاسخ از Response که ما فراهم کردیم را انجام می دهد.
متد call مقدار صفر یا یک را بر می گرداند که آیا فراخوانی سرویس با موفقیت پایان یافته است یا نه. عدم موفقیت ممکن ا ست در نتیجه ی م شکلاتی با زیر ساختهای رآس (برای مثال، تلاش برای فراخوانی سرویس به و سیله ی هیچ نودی انجام ن شده با شد) یا به دلیل م شخص فقط برای یک سرویس به وجود بیاید. در این مثال، یک call ناموفق معمولاً نشان می دهد که لاک پشت دیگری با نام درخواست شده وجود دارد.

یک ا شتباه معمول عدم چک کردن مقدار برگ شتی از call ا ست. اگر فراخوانی سرویس ناموفق با شد، این کار ممکن ا ست م شکلات غیرمنتظره ای را به وجود بیاورد. چک کردن این مقدار و فراخوانی یکROS_ERROR_STREAM در صورت ناموفق بودن فراخوانی سرویس، فقط یک یا دو دقیقه طول می کشد. این کار مانند یک سرمایه گذاری روی زمان ا ست که ا شکال زدایی های بعدی را سریعتر می کند.

به صورت پیش فرض، پرو سه پیدا کردن و مت صل شدن به نود سرور درون متد call اتفاق می افتد. این اتصال برای فراخوانی سرویس استفاده می شود و قبل از بازگشت call بسته می شود.

رآس مفهومی به نام مشتری های مداوم سرویس دارد، که ساختار ros::ServiceClient ارتباط با سرور را برقرار می کند، که در همه ی call های بعدی برای آن شی ء مشتری ا ستفاده می شود. با فرستادن true به عنوان دومین پارامتر در ساختار می توان یک مشتری مداوم سرویس ایجاد کرد. (که ما در مثال قبلی آن را به حالت پیش فرض false باقی گذاشتیم.)

```
ros::ServiceClient client =  node_handle.advertise<service_type>(service_name,
true);
```

استفاده از مشتری مداوم تقریباً در اسناد توصیه نشده است.[1] چون افزایش عملکرد زیاد نیست (یک آزمایش غیررسمی نویسنده نشان دهنده تنها ۱۰٪ افزایش عملکرد است) درحالیکه سیستم نهایی در مقابل دوباره شروع کردن و تغییر در نود سرور کمتر مقاوم خواهد بود.

بعد از اینکه فراخوانی سرویس با موفقیت به پایان رسید، شما به داده های پاسخ از شیء درخواست (Request) در call دسترسی دارید. در این مثال، پاسخ تنها شامل قسمت name درخواست Request است.

اعلام وابستگی ها: تا اینجا تو ضیح دادیم نود مشتری را چگونه ایجاد کنیم. هرچند، برای اینکه catkin_make یک برنامه مشتری را درست کمپایل کند، ما باید مطمئن باشیم که پکیج برنامه وابستگی به پکیج تایپ سرویس را به خوبی مشخص کرده است. برای این وابستگی ها، که مانند نوع پیام ها بدانها نیاز داریم (بخش ۳,۳,۳ را ببینید)، باید CMakeLists.txt و package.xml را ویرایش کنیم. برای کمپایل مثال، باید مطمئن شویم که در خط find_package در فایل CMakeLists.txt به پکیج turtlesim اشاره شده است.

```
find_package(catkin REQUIRED COMPONENTS roscpp turtlesim)
```

و در package.xml باید مطمئن باشیم المان های build_depend و run_depend با اسم پکیج وجود دارند.

```
<build_depend>turtlesim</build_depend>
<run_depend>turtlesim</run_depend>
```

بعد از اضافه کردن این تغییرات، catkin_make باید بتواند برنامه را طبق معمول کمپایل کند.

[1] http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html

۸-۴- برنامه سرور

بیاید به طرف دیگر فراخوان سـرویس ها نگاهی بیاندازیم، با نوشـتن برنامه ای که به عنوان سـرور عمل می کند. لیسـت ۸,۲ سـرویس toggle_forward را تشـکیل می دهد و همچنین ربات turtlesim را می راند، بین حرکت به جلو و چرخیدن هر دفعه که سـرویس را فرا می خواند تغییر می کند.

برنامه برای عمل کردن به عنوان سرور تقریباً مشابه با برنامه برای گوش کردن به تاپیک هاست. به جزء تفاوت در اسـمشـان (که ما باید ros::ServiceServer را به جای ros::Subscriber اسـتفاده کنیم). تنها تفاوت این است که یک سرور می تواند داده ها را به مشتری برگرداند، به صورت پاسخ شی ء یا به صورت یک تک بیتی (بولین) که موفقیت یا عدم موفقیت را اعلام می کند.

نوشتن یک سرویس بازگشتی callback: درست مانند شنونده ها، هر سرویس که نود ما آن را ایجاد می کند باید به یک تابع callback نسبت داده شود. یک callback سرویس مانند زیر است:

```
bool function_name(
package_name::service_type::Request &req),
package_name::service_type::Response &resp)
){
...
}
```

رآس تابع callback را هر دفعه برای هر فراخوانی سرویس که نود ما دریافت می کند اجرا می کند. پارامتر Request شامل داده هایی است که از مشتری فرستاده شده است. وظیفه ی callback پر کردن داده های قسمت های شی ء Response است. تایپ های Request و Response مشابهی وجودی دارد که ما در طرف مشتری در قسمت قبل استفاده کردیم، مانند، آنها به سربرگ مشابه و وابسـتگی های مشـابه در پکیج برای کمپایل شـدن نیاز دارند. تابع callback باید مقدار true را برگرداند برا ی نشان دادن موفقیت و مقدار false را برای نشان دادن عدم موفقیت.

در این مثال، ما از تایپ std_srvs/Empty اسـتفاده می کنیم، که Request و Response هر دو طرف خالی هستند، در نتیجه پرداز شی برا ی انجام دادن وجود ندارد. callback فقط برای شروع یک متغیر جهانی بولین به نام forward اسـتفاده می شـود که انتشـار پیام سـرعت در main را تضمین می کند.

```
1  // This program toggles between rotatation and translation
2  // commands, based on calls to a service.
3  #include <ros/ros.h>
4  #include <std_srvs/Empty.h>
```

```
5  #include <geometry_msgs/Twist.h>
6
7  bool forward = true;
8  bool toggleForward(
9    std_srvs::Empty::Request &req,
10           std_srvs::Empty::Response &resp
11         ) {
12           forward = !forward;
13           ROS_INFO_STREAM("Now sending " << (forward ?
14             "forward" : "rotate") << " commands.");
15           return true;
16         }
17
18         int main(int argc, char **argv) {
19           ros::init(argc, argv, "pubvel_toggle");
20           ros::NodeHandle nh;
21
22           // Register our service with the master.
23           ros::ServiceServer server = nh.advertiseService(
24             "toggle_forward", &toggleForward);
25
26           // Publish commands, using the latest value for forward,
27           // until the node shuts down.
28           ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
29             "turtle1/cmd_vel", 1000);
30           ros::Rate rate(2);
31           while(ros::ok()) {
32             geometry_msgs::Twist msg;
33             msg.linear.x = forward ? 1.0 : 0.0;
34             msg.angular.z = forward ? 0.0 : 1.0;
35             pub.publish(msg);
36             ros::spinOnce();
37             rate.sleep();
38           }
39         }
```

لیست (۸-۲) برنامه ی pubvel_toggle.cpp که دستورات سرعت که منتشر می کند را براساس سرویسی که درخواست می کند تغییر می دهد.

ساختن یک شی ء سرور: برای نسبت دادن تابع callback به ا سم سرویس، و برای فر ستادن سرویس به نودهای دیگر، ما باید سرویس را منتشر کنیم:

```
ros::ServiceServer server = node_handle.advertiseService(
service_name,
pointer_to_callback_function
);
```

همه ی این المان ها قبلاً ظاهر شده اند:

- Node_handle همان نگهدارنده ی نود است که ما می دانیم.
- Service_name نام ر شته ای سرویسی ا ست که می خواهیم درخوا ست دهیم. این ا سم باید (از نظر منطقی) اسم نسبی باشد، اما می تواند اسم جهانی هم باشد.

به خاطر بعضی از ابـ ها مات در مورد چگو نه اسـت فاده کردن از اسم خصوصی، ros::NodeHandle::advertiseService از پذیرفتن ا سم خ صو صی امتناع می کند. (ا سمهایی که با ~ شروع می شود.) راه حل این محدودیت ایجاد شی ء ros::NodeHandle با فضای نامی خاص خودش است. برای مثال- ما باید ros::NodeHandle مانند زیر ایجاد کنیم:

```
ros::NodeHandle nhPrivate(" ~ ");
```

فضای نامی پیش فرض برای هر نام نسبی که به NodeHandle فرستادیم مانند اسم نود خواهد بود. در عمل، این بدین معنا ست که اگر از این نگهدارنده و اسم نسبی برای منتشر کردن یک سرویس ا ستفاده کنیم، این همان تأثیر ا ستفاده از ا سم خاص را خواهد گذا شت. برای مثال، در نودی به نام/foo/bar ما می توانیم سرویسی به نام /foo/bar/baz را منتشر کنیم:

```
ros::ServiceServer server = nhPrivate.advertiseService(
"baz",
callback
);
```

این کد همان تأثیر که ما ممکن اسـت برای منتشـر کردن سـرویس به نام baz~ با اسـتفاده از NodeHandle معمول انتظار داریم را خواهد دا شت، اگر آن نگهدارنده ا سم های خاص را قبول می کرد.

- آخرین پارامتر، یک ا شارگر به تابع callback ا ست. یک معرفی سریع از ا شاره گر توابع، شامل چند پی شنهاد در مورد خطاهای احتمالی، در صفحه ۵۶ آورده شده ا ست. ایده ی مشابه اینجا اجرا می شود.

مانند شیء ros::Subscriber، به ندرت متدهای شی ء ros::ServiceServer را فرا می خوانیم. در عوض باید مدت طول عمر شـی ء را در نظر بگیریم، چون سـرویس برای نودهای دیگر باید فقط تا زمانی که ros::ServiceServer از بین می رود در دسترس باشد.

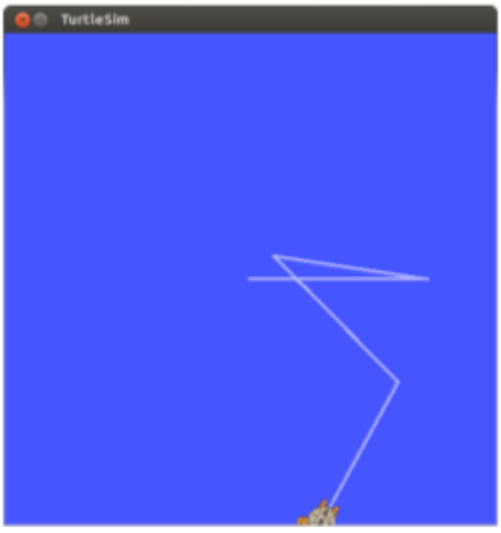
دادن کنترل به رآس : فراموش نکنید که رآس توابع callback را تا زمانی که ما مشـــخصـــاً درخواســـت، با اســـتفاده از ()ros::spin و()ros::spinOnce نکنیم، اجرا نمی کند. (جزیئات تفاوت بین این دو تابع در متن برنامه شنونده، نزدیک آخر بخش ۳,۴ مشخص است.)
در این مثال، ما از ()ros::spinOnce، به جای ()ros::spin اســـتفاده می کنیم، چون ما کار دیگری برای انجام دادن داریم(مشخصا منتشر کردن دستورات سرعت) وقتی ورودی از فراخوان سرویس برای پردازش نداریم.

۸-۴-۱- اجرا کردن و بهبود بخشیدن برنامه سرور

برای تســـــت برنـامـه pubvel_toggle، آن را کمپـایـل کنیـد و برنـامـه turtlesim_node و pubvel_toggle را اجرا کنید. با اجرای این دو برنامه، شما می توانید با استفاده از toggle_forward دستور حرکت را از سرعت به چرخش یا برعکس تغییر دهید:

```
rosservice call /toggle_forward
```

شکل ۸,۱ مثالی از نتیجه را نشان می دهد.



شکل (۸-۱) نتیجه ی اجرای pubvel_toggle با مقداری چرخش، با فراخواندن دستی toggle_forward/.

یک حالت غیرمنتظره ممکن در این برنامه یک گپ بین شـروع دسـتور و مشـاهده تغییر واقعی در حرکت لاک پشت وجود داشته باشد. یک قسمت بسیار کوچک این تأخیر به دلیل زمان مورد نیاز

برای ارتباط بین rosservice call، pubvel_toggle ، و turtlesim_node است. به هر حال، بیشتر این تأخیر از ساختار pubvel_toggle ناشی می شود. می توانید ببینید کجا؟
جواب این است که، ما از متد sleep از شیء ros::Rate با فرکانس تقریباً پایینی (فقط ۲ هرتز) استفاده می کنیم، این برنامه بیشتر وقتها در حالت خواب (انتظار) است. بیشتر فراخوانی سرویس ها بعد از اجرای sleep می رسند، و این فراخوانی سرویس نمی تواند تا خط ros::spinOnce() اجرا شود، که فقط هر ۰٫۵ ثانیه اتفاق می افتد. بنابراین، قبل از اجرای هر فراخوانی سرویس ممکن است ۰٫۵ ثانیه تأخیر وجود داشته باشد.
حداقل دو راه حل برای این مشکل وجود دارد:

- ما می توانیم دو thread جداگانه استفاده کنیم: یکی برای منتشر کردن پیام ها، و یکی دیگر برای اجرای تابع callback سرویس. گرچه رآس به برنامه هایی برای استفاده از thread به صورت صریح نیاز ندارد، اگر این کار را انجام بدهند کاملا متوافق خواهد بود .
- ما می توانیم حلقه ros::spinOnce / sleep را با ros::spin عوض کنیم، و از timer callback برای منتشر کردن پیام ها استفاده کنیم.

مواردی مانند این می توانند در این مقیاس فعلی کوچک به نظر بیایند (یک تأخیر کوچک در جهت سر لاک پشت مشکل بزرگی نیست) اما برای برنامه هایی که زمان بسیار مهم است، اختلاف می تواند حیاتی باشد.

۸-۵- در ادامه

در این فصل سرویس ها را پوشش دادیم، که شباهت های بسیاری و تفاوتهای حیاتی با پیام ها داشتند. در فصل بعدی به جهت دیگری می رویم، و ابزاری به نام rosbag را یاد می گیریم، که یک آزمایش سریع و تکراری را با ذخیره کردن و بازاجرا کردن پیام ها ممکن می کند.

فـصل ۹ : ضبط و بازپخش پیام ها

در این فصل ما یاد می گیریم چگونه از فایل های *bag* برای ضبط کردن و بازپخش کردن پیام ها استفاده کنیم.

یکی از ویژگی های اولیه ی سیستم خوب طراحی شده ی رآس این است که قسمتی از سیستمی که از اطلاعات استفاده می کند، نباید در مورد مکانیسم ایجاد شدن اطلاعات آگاه باشد. این ساختار را می توان به راحتی در مدل ارتباطات ناشر-شنونده رآس دید. یک نود شنونده خوب باید هر زمان که پیام های مورد نیازش منتشر شدند کار کند، بدون توجه به اینکه چه نود یا نودهایی آن پیام ها را منتشر کرده اند.
این بخش ابزاری به نام rosbag را تو صیف می کند که یک مثال واقعی از این نوع انعطاف ا ست. با rosbag می توانیم پیام های منتشر شده روی یک یا چند تاپیک را درون یک فایل ذخیره کنیم، و بعداً می توانیم این پیام ها را بازپخش کنیم. با این دو قابلیت یک روش قدرتمند برای تست نرم افزار بعضی رباتها ایجاد کرده ایم: می توانیم ربات را فقط برای زمان های کوتاهی اجرا کنیم، تاپیک های مورد نظر را ذخیره کنیم، و این پیام ها روی این تاپیک ها بارها بازپخش کنیم، و نرم افزاری که این داده را پردازش می کند را تست کنیم.

۹-۱- ذخیره کردن و دوباره پخش کردن فایل های bag

کلمه فایل bag به فایلی با فرمت خاص اشاره می کند که پیام های رآس همراه با برچسب های زمانی را ذخیره می کند. دستور rosbag می تواند برای ذخیره و بازپخش فایل های bag استفاده شود.[۱۲]

ذخیره کردن فایل های bag : برای ایجاد فایل bag، از دستور rosbag استفاده کنید:
rosbag record -O filename.bag topic-names
اگر نام فایل را مشخص نکنید، rosbag خودش نامی براساس تاریخ و زمان انتخاب خواهد کرد. بعلاوه، چند مورد دیگر وجود دارد که می تواند برای rosbag record مفید باشد.

- به جای لیست کردن تاپیک های مشخص، شما می توانید با استفاده از rosbag record -a پیام های همه ی تاپیک هایی که در حال حاضر منتشر شده اند را ذخیره کنید.

[۱] http://wiki.ros.org/rosbag
[۲] http://wiki.ros.org/rosbag/Commandline

ذخیره کردن همه ی تاپیک ها مشکلی برای سیستمی با ابعاد کوچک (که در این کتاب وجود دارد) ایجاد نمی کند. هرچند، این می تواند ایده ی بدی در سیستم ربات های واقعی باشد. برای مثال، بیشتر ربات ها دارای دوربین، نودی دارند که چندین تاپیک شامل تصاویر با پردازش مختلف و در سطوح مختلف متراکم شده را منتشر می کنند. ذخیره کردن همه ی این تاپیک ها به سرعت فایل های bag بزرگی را ایجاد می کند. وقتی می خواهید از a- استفاده کنید تجدید نظر کنید، یا حداقل اندازه فایل bag را در نظر بگیرید.

- شما می توانید فشرده سازی را درون فایل bag با j- rosbag record فعال کنید. این دستور مصالحه معمول در فشرده سازی را دارد: به طور کلی اندازه فایل کوچکتر در مقابل مقدار بیشتری محاسبه حین خواندن و نوشتن. به نظر نویسنده، فشرده سازی به طور معمول ایده ی خوبی برای فایل های bag به نظر می رسد.

وقتی ذخیره سازی تمام شد، می توانید از Ctrl-C برای متوقف کردن rosbag استفاده کنید.

بازپخش کردن فایل های bag: برای بازپخش کردن یک فایل bag، از دستور زیر استفاده کنید:

```
rosbag play filename.bag
```

پیام هایی ذخیره شده در فایل bag بازپخش می شوند، به همان ترتیب و با همان تناوب زمانی بینشان که از در اصل منتشر شده بودند.

بازرسی فایل های bag: دستور rosbag info اطلاعات جالبی را در مورد یک فایل bag فراهم می کند:

```
rosbag info filename.bag
```

به عنوان مثال، اینجا خروجی در مورد یک فایل bag که نویسنده هنگام نوشتن بخش بعدی ذخیره کرده است می بینید:

```
path: square.bag
version: 2.0
duration: 1:08s (68s)
start: Jan 06 2014 00:05:34.66 (1388984734.66)
end: Jan 06 2014 00:06:42.99 (1388984802.99)
size: 770.8 KB
messages: 8518
compression: none [1/1 chunks]
types:  geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
turtlesim/Pose [863b248d5016ca62ea2e895ae5265cf9]
topics: /turtle1/cmd_vel 4249 msgs : geometry_msgs/Twist
/turtle1/pose 4269 msgs : turtlesim/Pose
```

بخصوص مدت (duration)، تعداد پیام ها (messages)، لیست تاپیک ها بسیار جالب هستند.

۹-۲- مثال: یک bag از مربع ها

بیاید با یک مثال ببینیم که چگونه فایل های bag کار می کنند.

ترسیم مربع ها: ابتدا، roscore و نود turtlesim_node اجرا کنید. از پکیج turtlesim، نود draw_square را اجرا کنید:

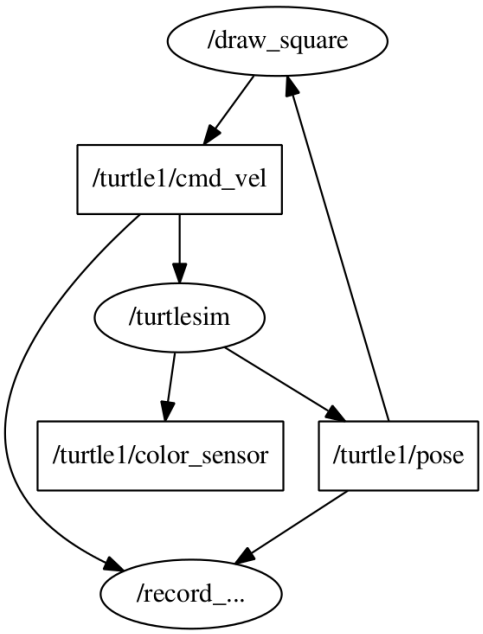
```
rosrun turtlesim draw_square
```

این نود شبیه ساز را دوباره شروع می کند (با فراخوانی سرویس reset) و دستور سرعت را منتشر می کند که لاک پشت را در یک الگویی تقریباً مربع شکل حرکت می دهد. (شما همچنین می توانید از نودهایی که ما نوشته ایم و دستور سرعت را منتشر کردیم استفاده کنید. برنامه ی draw_square ترجیح دارد چون برعکس pubvel می توان به سادگی ساختار حرکتی لاک پشت را دید.)

ذخیره فایل bag مربع ها: همان طور که لاک پشت مربع ها را می کشد، دستور زیر را برای ذخیره کردن دستورات سرعت و موقعیت لاک پشت اجرا کنید:

```
rosbag record -O square.bag /turtle1/cmd_vel /turtle1/pose
```

خروجی اولیه ی نشان می دهد که rosbag به /turtle1/cmd_vel و /turtle1/pose گوش می کند، و آنها را در square.bag ذخیره می کند. در این جا، گراف (نشان داده شده به وسیله ی rqt_graph) به مانند شکل ۹,۱ است. قسمت جدید و جالب این است که rosbag یک نود جدید به نام /record_ . . . ، که به /turtle1/cmd_vel گوش می دهد. گراف نشان می دهد که rosbag به وسیله ی گوش دادن به تاپیک هایی که شما درخواست کرده اید ،مانند نود های دیگر، با استفاده از مکانیسم مشابه که در بخش ۳ یاد گرفته ایم، پیام ها را ذخیره می کند.



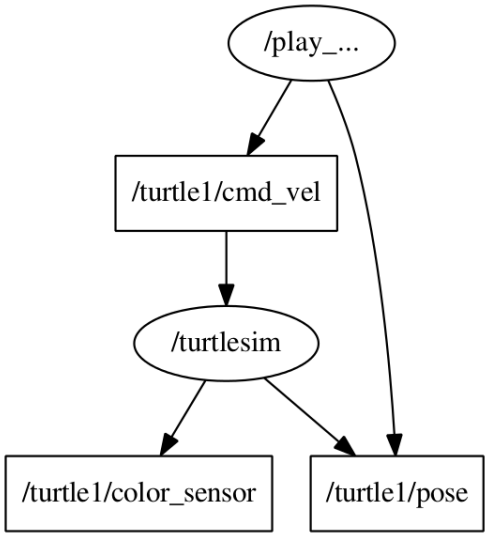
شکل (۹-۱) گراف نودها و تاپیک ها در حین اجرای rosbag record .

نودهایی که rosbag ایجاد می کند از نام های مستعار که در بخش ۵,۴ تو ضیح دادیم ا ستفاده می کند. در این بخش، ما این اعداد دنبال نام را به وسیله سه نقطه (. . .) جایگزین کردیم. توجه داشته باشید که استفاده از نام مستعار یعنی اینکه می توانیم همزمان چند rosbag record را با هم اجرا کنیم.

بازپخش کردن فایل bag **مربع ها:** بعد از اجرای سیستم برای مدتی ـــ یک یا دو دقیقه زیاد هم هســـت - rosbag را متوقف کنید تا ذخیره کردن را متوقف کنید و نود draw_square را متوقف کنید تا لاک پشـــت بایســـتد. بعد، bag را بازپخش کنید. بعد از اینکه مطمئن شـــدید roscore و turtlesim هنوز اجرا می شوند، از دستور زیر استفاده کنید:

rosbag play square.bag

توجه داشته باشید که لاک پشت به حرکت کردن ادامه می دهد. این اتفاق می افتد چون rosbag نودی به نام play_. . . را ایجاد می کند که turtle1/cmd_vel/ را منتشـــر می کند، همان طور که در شکل ۹,۲ نشان داده شده است. همان طور که ما انتظار داریم، پیام هایی که منتشر شده اند به مانند پیام های منتشر شده به وسیله ی draw_square هستند.



شکل (۹-۲) گرافی از نودها و تاپیک ها حین اجرای rosbag play

شـکل ۹,۳ نقش هایی از نتیجه ی دنباله ای از این عمل را نشـان می دهد. براسـاس اینکه شـما چگونه در مورد عملکرد rosbag فکر کرده اید، نقش ها می تواند تعجب برانگیز باشند.

شکل (۹-۳) (چپ) یک لاک پشت در پاسخ به دستور حرکت از draw_square. حرکت ها به وسیله ی rosbag ذخیره شده است. (راست) با بازپخش کردن فایل bag, ما دنباله ای از پیام ها را برای لاک پشت می فرستیم.

- مربع های ذخیره شـده در طول rosbag ممکن اسـت در همان مکان مربع ها در حین rosbag record کشیده نشوند. چرا کشیده نمی شوند؟ چون rosbag فقط دنباله ی پیام ها را تکرار می کند. این د ستور موقعیت اولیه را تکرار نمی کند. دومین دنباله ی مربع ها، در حین rosbag play، از جایی شروع می شود که لاک پشت هست.

- Draw_square و rosbag play کمی توانند لاک پشت را به جای مختلفی بفرستند، حتی اگر bag شامل داده موقعیت تاپیک turtle1/pose/ نیز باشد. چرا؟ خیلی ساده، چون در این مثال، هیچ کس دیگری به جز rosbag record به turtle1/pose/ گوش نمی دهد. اینجا تفاوتی بین محل لاک پشت که نودی (در اینجا rosbag play) داده اش را منتشر می کند و جایی که واقعاً لاک پشت هست وجود دارد. داده موقعیت از طرف فایل bag در نظر گرفته نمی شود.

در واقع وقتی هر دو turtlesim_node و rosbag play اجرا می شوند، پیام ها روی turtle1/pose/ می توانند کاملاً متضاد باشند. لیست ۹,۱ مثالی از چهار پیام منتشر شده روی این تاپیک، در کمتر از یک ثانیه، را نشان می دهد. به تغییرات سریع روی مختصات y توجه کنید. خوشبختانه هیچ نودی به این تاپیک گوش نمی دهد، چون چنین نودی برای درک این پیام ها دچار مشکل می شود.

```
1   x : 5.93630695343
2   y : 4.66894054413
3   theta : 5.85922956467
4   linear_velocity : 0.0
5   angular_velocity : 0.40000000596
6   ---
7   x : 5.56227588654
8   y : 7.4833817482
9   theta : 4.17920017242
10          linear_velocity : 0.0
11          angular_velocity : 0.40000000596
12          ---
13          x : 5.93630695343
14          y : 4.66894054413
15          theta : 5.865629673
16          linear_velocity : 0.0
17          angular_velocity : 0.40000000596
18          ---
19          x : 5.56227588654
20          y : 7.4833817482
21          theta : 4.18560028076
22          linear_velocity : 0.0
23          angular_velocity : 0.4000000059
```

لیست (۹-۱) برنامه ی pubvel_toggle.cpp که دستورات سرعت که منتشر می کند را براساس سرویسی که درخواست می کند تغییر می دهد.

یاد بگیرید که از سیستم هایی که هر دوی rosbag و نودهای واقعی را روی یک تاپیک منتشر می کنند پرهیز کنید. (یا حداقل در موردشان خیلی احتیاط کنید.)

- شکل ۹,۳ مشخص می کند که فراخوان های سرویس (فصل ۸) درون فایل های bag ذخیره نشده است. اگر آنها ذخیره می شدند، bag باید شامل ذخیره هایی می بود از زمانی که draw_square سرویس /reset را قبل از شروع فرستادن پیام ها فرا می خواند، و لاک پشت به سر جای اولش باز می گشت.

۹-۳- Bags درون فایل های لانچ

به جزء دستور rosbag که تا حالا دید، رآس نود های قابل اجرایی به نام record و play فراهم کرده است که قسمتی از پکیج rosbag هستند. این برنامه ها عملکرد مشابه ای دارند و به ترتیب پارامترهای مشابه ای در کامند لاین های به عنوان rosbag record و rosbag play می پذیرند.
این بدان معنا است که این امکان وجود دارد (اما لازم نیست) که فایل های bag به وسیله rosrun ذخیره و بازپخش شوند، مانند زیر:

```
rosrun rosbag record -O filename.bag topic-names
rosrun rosbag play filename.bag
```

مهمتر از همه، نود های قابل اجرای record و play اضافه کردن فایل های bag به فایل های لانچ را، با اضافه کردن المان های نود مناسب، راحتتر می کنند. برای مثال یک نود record می تواند به صورت زیر باشد:

```
<node
pkg="rosbag"
name="record"
type="record"
args="-O filename.bag topic-names"
/>
```

و نود play می تواند به صورت زیر باشد:

```
<node
pkg="rosbag"
name="play"
type="play"
args="filename.bag"
/>
```

به جزء نیاز به args در دستورشان، این نودها به هیچ چیز غیر معمول دیگری در فایل لانچشان نیاز ندارند.

در اینجا ممکن است از اینکه این فصل بدون بحث در مورد چگونه استفاده از فایل های bag در C++ به پایان رسید متعجب شده باشید. در واقع یک API برای خواندن و نوشتن فایل های bag وجود دارد.[1] هرچند، این API واقعاً فقط برای کاربردهای خاص مورد نیاز است. برای ذخیره کردن و بازپخش های عادی، دستور rosbag کاملاً کافی است.

۹-۴- در ادامه

در اینجا گذر ما بر اجزای اساسی رآس به پایان می رسد. در فصل بعدی با ذکر مختصری از تعدادی دیگر از موضوعاتی که در سیستم های واقعی رآس ظاهر می شوند کتاب به پایان می رسد.

[1] http://wiki.ros.org/rosbag/CodeAPI

فـصل ۱۰ : جمع‌بندی

در این فصل ما چند تاپیک دیگر را نشان می دهیم.

در فصل های قبل، ما به کارهای اصلی رآس را با جزئیات نگاهی انداختیم. ما مثال هایی از بیشتر مفاهیم با استفاده از شبیه ساز turtlesim دیدیم. به طور حتم، انگیزه اصلی شما برای رآس مشابه به حرکت در آوردن یک لاک پشت نیست. اگر این کتاب کارش را به خوبی انجام داده باشد، شما باید برای استفاده از رآس برای حل کردن مسائل رباتیکی مورد نظرتان آماده باشید، به خصوص با استفاده از اسناد کمک آموزشی، و با استفاده از قسمت کمک آموزشی پکیج های موجود.

۱۰-۱- قدم بعدی

این فصل شامل نوشته های خلاصه ای (با ارجاع به اسناد مربوطه) در مورد تعداد تاپیک متداول در سیستم واقعی رآس است که البته ما آنها را تا اینجا پوشش ندادیم.

اجرای رآس در شبکه: شما ممکن است از فصل ۱ به خاطر بیاورید که یکی از امتیازات رآس این است که تسهیلات توزیع عملکرد ربات ها را فراهم می کند، در مواردی که برنامه های متفاوتی در چندین کامپیوتر اجرا شوند، می توانند با یکدیگر ارتباط برقرار کنند. هرچند، در طول این کتاب، کل سیستم رآس درون یک کامپیوتر اجرا شد.

برای استفاده از رآس درون شبکه ای با چندین کامپیوتر، نیازمند تنظیم کردن در هر دو سطح شبکه (برای اینکه مطمئن شویم کامپیوترها می توانند با هم صحبت کنند) و سطح رآس (که مطمئن شویم همه ی نودها می توانند با مستر ارتباط برقرار کنند.) هستیم.[۱ ۲ ۳] خبر خوب این است که، وقتی شما این موارد را درست تنظیم کردید، رآس جزئیات ارتباط شبکه را تحت نظر می گیرد. نودها درون کامپیوترهای مختلف به همان صورت ارتباط برقرار می کنند که ما در یک کامپیوتر استفاده می کردیم.

نوشتن یک برنامه تمیز: سورس کدهای مثال های این کتاب اساساً برای اختصار و واضح بودن بهینه شده بودند، نه برای گسترش و قابل نگهداری بودن. در واقع، راهنمایی های زیادی برای

[۱] http://wiki.ros.org/ROS/NetworkSetup

[۲] http://wiki.ros.org/ROS/Tutorials/MultipleMachines

[۳] http://wiki.ros.org/ROS/EnvironmentVariables

نوشتن یک برنامه ی تمیز پیشنهاد شده است که ما در این کتاب رعایت نکردیم. برای مثال، بعضی از گسترش دهنده ها پیشنهاد کرده اند که از ros::Timer به جای ros::Rate استفاده کنیم.[۱] بعضی از گسترش دهنده ها همچنین ترجیح می دهند با استفاده از قرار داردن همه ی قسمت ها یا قسمتی از داده ای نود درون یک کلاس، و استفاده از متدهای آن کلاس به عنوان توابع های فراخوان شده تعداد متغیرها و توابع جهانی را کاهش دهیم.[۲] توجه به این نوع تکنیک ها سایز و پیچیدگی برنامه ها را زیاد می کند.

نشان دادن داده ها با استفاده از RVIZ : این ابزار نیز با لاکپشت کار می کند. تقریبا همه داده ها در پیام های ما با اطلاعات ساده ای همچون موقعیت های دو بعدی، جهت ها و سرعت ها هستند. در مقابل در ربات های واقعی داده ها اغلب بسیار پیچیده تر می باشند و هیچ کدام از تکنیک هایی که ما در این کتاب بکار گرفتیم واقعا مناسب دیدن و محاسبه داده های سنگین و پیچیده و نویزی که آنها تولید می کنند نمی باشد. برای پر کردن این شکاف رآس ابزار گرافیکی به نام rviz که می تواند گستره ی وسیعی از داده ها را نمایش دهد (طبیعتا با سابسکرایب کردن به موضوعات انتخابی کاربر) فراهم کرده است[۳].

ساختن انواع پیام ها و سرویس ها: پیام ها و سرویس های مربوط به مثال این کتاب همگی منحصرا بر اساس پیام ها و سرویس ها موجود بود. با این حال ساختن انواع پیام ها و سرویس هایی که متعلق به پکیج خودمان باشد کار سر راستی است[۴،۵].

مدیریت چارچوب های مختصات با tf : از آنجایی که ربات ها در دنیای واقعی کار می کنند، بسیار طبیعی است که موقعیت قسمت های مختلف ربات و اشیایی که ربات باید با آنها در ارتباط باشد یا از آنها بپرهیزد، در چارچوب های مختصات نشان داده شود. بنابراین مدیریت چارچوب های مختصات بسیار حیاتی است. انواع پیام های بسیاری شامل frame_id وجود دارد که چارچوب مختصاتی که در آن داده های پیام ها بیان می شود را شناسایی می کند. برای استفاده از چارچوب ها ما باید رابطه ی آنها را با همدیگر بدانیم. به خصوص ما اغلب مایلییم **تبدیل** بین این چارچوب ها را بدانیم. رآس پکیج های استانداردی به نام tf را برای این کار فراهم آورده است. پکیج های tf

---

۱ http://wiki.ros.org/roscpp/Overview/Timers

۲ http://wiki.ros.org/roscpp_tutorials/Tutorials/UsingClassMethodsAsCallbacks

۳ http://wiki.ros.org/rviz

۴ http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv

۵ http://wiki.ros.org/ROS/Tutorials/DefiningCustomMessages

به صورت مقاوم طراحی شده اند به طوری که اگر اطلاعات مربوطه در نودهای مختلفی باشند و حتی وقتی این اطلاعات متغیر با زمان باشند، کار می کنند[۱][۲][۳][۴].

شبیه سازی با Gazebo : یکی از بزرگترین مزایای طراحی ماژولار نرم افزار که رآس مشوق آن است، این است که ما به راحتی می توانیم اجزای مختلف یک سیستم را از هم جدا نماییم تا بدین وسیله زمان پیاده سازی را کوتاه و تست سیستم را راحت تر نماییم. فصل ۹ مثالی از این قابلیت را بیان نمود که در آن به صورت موقت یک یا چند نود از سیستم را با فایل های bag از پیام های از پیش ضبط شده ای جایگزین نماییم. یک ابزار قدرتمند تر دیگر Gazebo می باشد که شبیه ساز ربات با قابلیت اتکای بالایی می باشد. با استفاده از Gazebo می توان مشخصات ربات و محیط را تعریف نمود و با ربات از طریق رآس همان گونه ارتباط برقرار کرد که با چیزهای واقعی ارتباط برقرار می کنیم[۵].

۱۰-۲- نگاهی به آینده

دراینجابه انتهای مقدمه آرام خود بر رآس می رسد. نویسنده صمیمانه امیدوار است که این نقطه ابتدای شروع استفاده از رآس برای خلق ربات هایی قویتر و کاراتر باشد.

---

۱ http://wiki.ros.org/tf/Tutorials/Introductiontotf

۲ http://wiki.ros.org/tf/Tutorials/Writingatflistener(C++)

۳ http://wiki.ros.org/tf/Overview/DataTypes

٤ http://ros.org/doc/indigo/api/tf/html/c++/

٥ http://gazebosim.org/wiki/Tutorials/1.9/Overview_of_new_ROS_integration