MESSAGE LOGIC IN ROBOTICS SYSTEMS

Students' names:
Gurshabad Grover, Shivam Thukral

Roll numbers: 2013038, 2013095

BTP report submitted in partial fulfillment of the requirements
for the Degree of B.Tech. in Computer Science & Engineering
on 18th July, 2016

BTP Track: Engineering

BTP Advisors:
Dr. Rahul Purandare
Dr. P. B. Sujit

Indraprastha Institute of Information Technology
New Delhi

Student's Declaration

We hereby declare that the work presented in the report entitled *Resolving Message Logic Dependencies in Robotics Systems* submitted by us for the partial fulfillment of the requirements for the degree of Bachelor of Technology in Computer Science & Engineering at Indraprastha Institute of Information Technology, Delhi, is an authentic record of our work carried out under guidance of Dr. Rahul Purandare and Dr. P. B. Sujit. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

.... Place & Date: .....

Gurshabad Grover Shivam Thukral

Certificate

This is to certify that the above statement made by the candidates is correct and to the best of my knowledge.

Dr. Rahul Purandare Dr. P.B. Sujit Place and Date

Abstract

Robotic systems are event-based systems focusing at achieving high degrees of autonomy. The final action taken is achieved through an accumulation of series of computations by independent components in the system which can communicate with each other by exchanging messages through channels.

State-of-the-art robotics frameworks provide a high-level architecture for message communication so that the actual mechanism is hidden from the end user. This is implemented through a system of publishers (message sources) and subscribers (message sinks) for communication.

Since each node has the ability to subscribe or publish to any channel, there is no direct lookup to see which components of a system are affected by changes in message definition or message logic. By exploring previous work and introducing an efficient method of program analysis, we worked towards a tool which can accurately resolve such message logic dependencies in popular frameworks for robot software development.

The communication between nodes relies on the messages being exchanged between them, forming a network of publishers and subscribers. The performance of the network can be defined in terms of latency and/or message drops. We investigate how the message size affects the performance of the network.

For any given field in a message, it may or may not be utilised by a node. If it is not used by any node in the system, we could prune that field and expect to see a rise in performance of the network. We performed tests and worked towards a tool which automatically detects unused fields and prunes them.

We also explored several other potential issues which could affect pace of development, or cause unexpected crashes in the system.

Keywords: Robotics system, messages, publisher, subscriber, network performance

Acknowledgements

This work could not have been possible without the immeasurable support and guidance of our advisors, Dr. Rahul Purandare and Dr. P. B. Sujit. We would also like to thank Dr. Sebastian Elbaum (UNL) and Nishant Sharma (UNL) for their continuous suggestions and feedback which have improved our work on this project.

Work Distribution

Both of us worked on:

- worked on the algorithm and proposal to prune messages fields from bloated messages
- performed manual implementation of the algorithms and different approaches to the bloated messages problem

Additionally,

Gurshabad:

- performed tests for the latency/message drop performance of bloated messages
- developed utility to create lighter ROS messages using appropriate information

Shivam:

- worked on the Clang tool to implement field usage detection in subscribers
- modified the tool made by Dr. Purandare and added support for more statements

CONTENTS

1. Introduction

Frameworks for robot software development provide a high level of hardware and software abstraction for common functionalities in robots. Efficient message handling and correct communication is crucial to any such event-based system, and popular frameworks like the Robot Operating System[1], and MOOS-IvP[2] provide a publisher-subscriber model for communication between components.

Communication between components is achieved through the transmission and receipt of messages. In a publisher-subscriber model, all components either publish to a communication channel (publishers), subscribe to channels to access messages (subscribers) on it or function as both.

These messages which facilitate communication are well-defined and consist of a set of fields. If a message field is not used by any node in the system, we could prune that field and expect to see a rise in performance of the network. We performed tests and worked towards a tool which automatically detects unused fields and prunes them.

This report also presents an analysis of bug reports we studied, and the solutions to some of the problems we encountered, namely:

1. Ones related to change in the logic of publishing a message
2. Ones related to updating of ROS on systems
3. Inconsistency issues in publisher/subscriber

In this report, each problem is covered in detail in its section with our contributions in identifying the best solution to the problem and its implementation.

1.1 Robot Operating System

The Robot Operating System (ROS) is a popular open-source set of software libraries and tools that help you build robot applications [1]. It provides a publisher-subscriber platform wherein components can exchange messages.

When a ROS package is running successfully, we can get the running processes and their underlying dependencies by monitoring the flow of messages. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive or post messages [3]. `rxgraph` (a tool included with ROS) display a visualization of a ROS Computation Graph, i.e. the ROS nodes that are currently running, as well as the ROS topics (communication channels) that connect them [4].

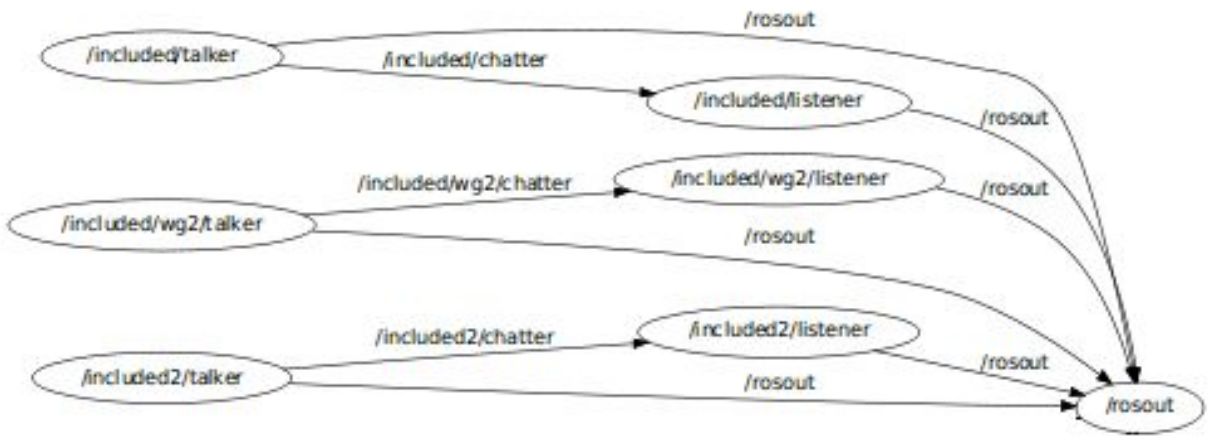


Figure 1: Example output of `rxgraph` which displays running nodes & topics that connect them [4]

1.2 Messages in ROS

Messages, in ROS, are specified using a simple descriptive language which is stored in a `.msg` file. According to this specification, ROS generates the source code for different target machines. Publishers in the system can make use of these message definitions and publish an instance of the message to communication channels.

```
int32 xPosition
int32 yPosition
string name
```

`Header.msg`

Figure 2: Example of a ROS message

2. Pruning message fields

2.1 Problem Outline

Fields in a message may be unused by the subscriber. These fields, having no utility of their own, take up additional and unnecessary space in all the messages transmitted from the publisher to the subscriber. We would like to investigate whether the intuition that messages of less size would increase performance in the robot.

Some metrics which offer a good definition of network performance:

- **Latency**: the time difference between the time of the message being sent by the publisher and the time the subscriber receives the message
- **Percentage of messages dropped**: Bloated messages could also lead to a higher percentage of messages being dropped by the buffers on the publisher/subscriber.

We will investigate whether these metrics are affected by increase in the message size. And if so, present a method to prune those message fields which remain unused at the subscriber end. Since a message can be utilised as the mode of communication in multiple channels at the same time, an elegant solution is required so that the performance of the network can be maximally optimised while ensuring that the actual behaviour of the system remains the same.

2.2 Experiment Setup

We will use differently-sized messages to investigate how message size affects the performance of the network in a robotics system. We varied the number of fields in the messages between 5, 10, 15, 20, and 25 wherein each message field carried a kilobyte of string data.

Our setup consisted of two devices on the same network:

1. A Raspberry Pi functioning as the publisher of the defined message
2. A laptop functioning as the subscriber to these messages

These devices are connected to each other through a common router via standard ethernet cables. The figure below summaries the setup.

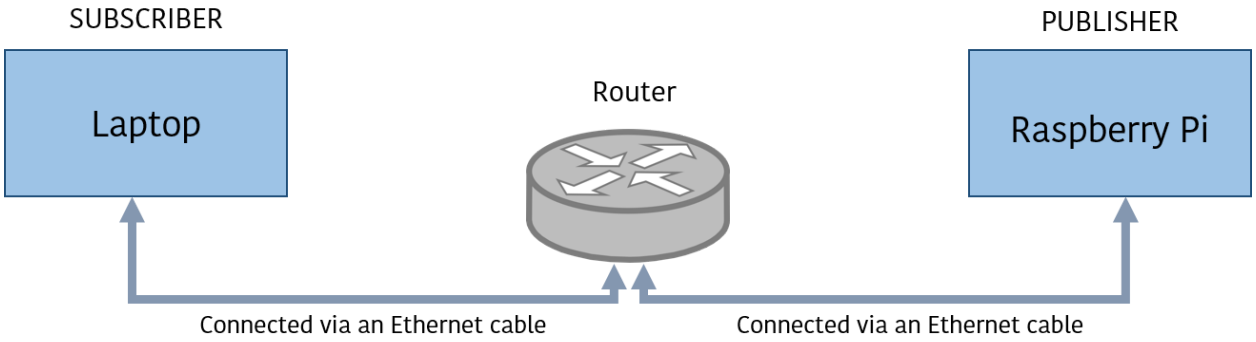


Figure 3: The setup of our network performance experiment

We defined custom messages to contain string fields, each of which the publisher was filling with a kilobyte-sized string. We varied the number of fields to simulate an increase in size of the message, implying that the total size of the message in kilobytes sent was numerically equal to the number of fields in the message.

- 25 fields
- 50 fields
- 100 fields

We repeated the experiment for each different type of message with another varying factor - the frequency at which the messages were published. The results in the end should establish a correlation with the network performance on both these varying factors (size and frequency). The frequencies at which we published these messages:

- 50 Hz
- 100 Hz
- 200 Hz

We measured the network performance based on two of characteristics, which have been defined in the previous section - latency in message transmission and percentage of messages dropped.

2.3 Results and Trends

The results are summarised in the figures below.

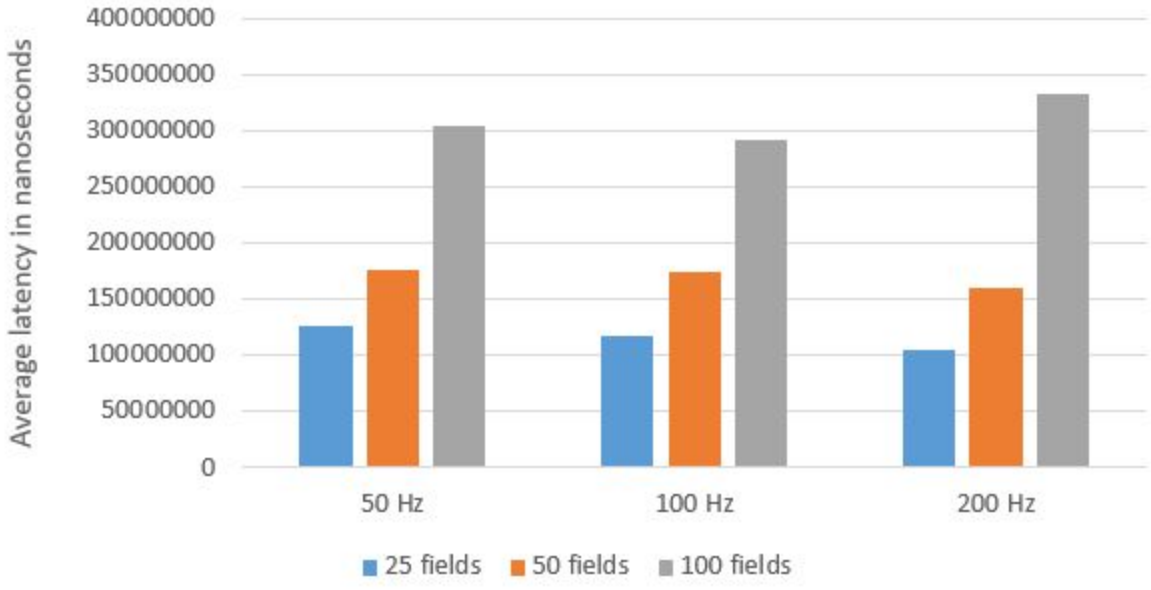


Figure 4: Trends in average latency in message transmission

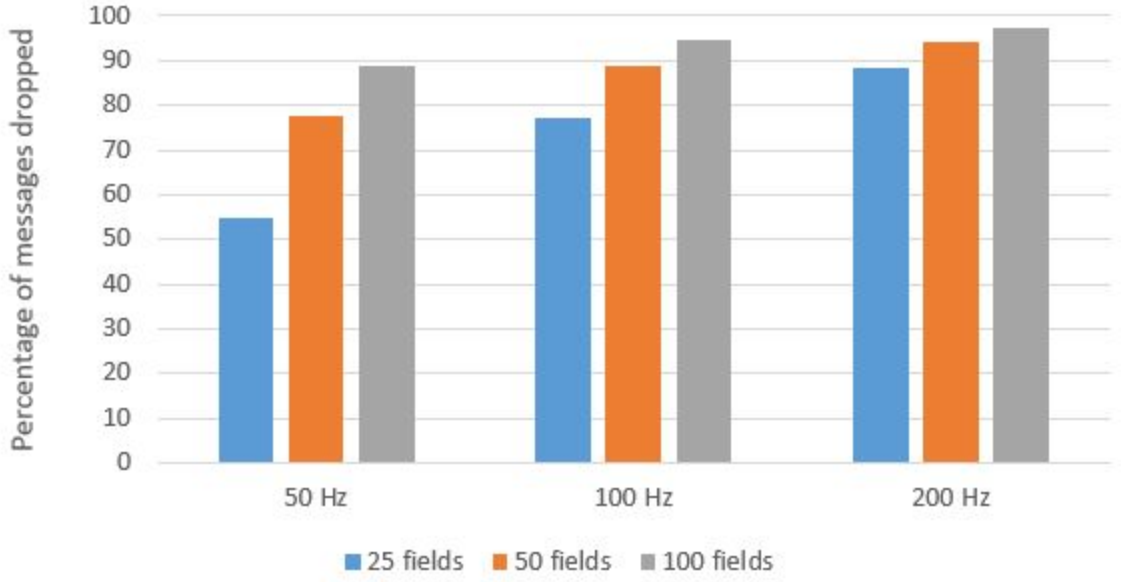


Figure 5: Trends in percentage of messages dropped

We can draw the following inferences from the trends in data:

- The percentage of messages dropped increases with an increase in message size.
- The percentage of messages dropped increases with an increase in the frequency at which messages are published.
- The average latency in message transmission increases with an increase in message size.
- The average latency in message transmission is insensitive to an increase in the frequency at which messages are published.

2.4 Overview of Proposed Solution

Problem statement:
Given the source code of a ROS system (may or may not contain multiple packages), identify and prune fields from defined messages, as per this table.

Input:
Source code of the ROS system, including:

- .cpp source files of publishers, subscribers
- .msg files containing message definitions

Output:
Modified source code of the ROS package, which now uses the pruned messages (without changing the behaviour of the code)
As required,

- New message files will be added
- The source code of publishers and subscribers will be modified to use new messages
- The CMakeLists.txt file will be modified

Algorithm Overview:

Our solution proceeds in four distinct phases, summarised in the figure below.

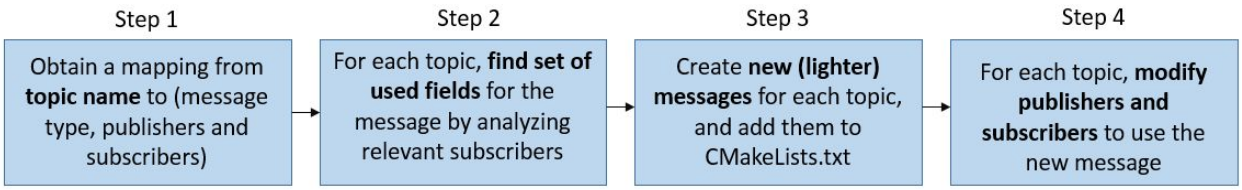


Figure 6: Overview of the proposed solution

2.5 Algorithm Pseudocode

```
/* STEP 1
To achieve the mapping, we will iterate through the source of all the executables of
the package. For each advertise statement found, we will associate the publisher to
the respective topic. For each subscribe statement found, we will associate the
subscriber to the respective topic.*/
            init topics_list
            For each executable in the CMakeLists folder:
                  If advertise statement is found with topic T:
                        Add <publisher_info, message> pair to topics_list[T]
                  If subscribe statement is found with topic T:
                        Add <subscriber_info> to topics_list[T]

/* STEP 2
For each topic, we will analyse the callback function of the subscribers subscribing
to the topic. If the use of a field is detected in this function, we will add it to
the ListOfUsedFields map.*/
            For each topic in topics_list:
                  ListOfUsedFields = {}
                  For each subscriber to the topic:
                        If use of a field is detected in the callback function:
                              Add field to ListOfUsedFields

/* STEP 3
For each topic, we will create a new message file with only the set of fields we
obtained in Step 2.*/
            For each topic in topics_list:
                  Create new message with fields from step 2
                  Add this message to CMakeLists
```

```
/* STEP 4
For each topic, we will modify its publishers and subscribers to use the new message.
This will be done through a namespace define method. */
            For each topic in topics_list:
               For each subscriber to the topic:
                   Remove line #include "pckg/old_msg.h"
                    Add line #include "pckg/new_msg.h"
                    Add line namespace pckg{typedef pckg::new_msg old_msg}

                For each publisher to the topic:
                     Add line #include "pckg/new_msg.h"
                     Modify advertise statement to use new message
                     Add statements to publish new message (*5)


/* STEP *5 - PUBLISHER MODIFICATION
This step creates an instance of the lighter message and publishes it on the channel
(instead of the older heavier message)
*/
            Find publish statement
            Remove publish statement
            Create an instance of the new message
            For each field in ListOfUsedFields:
                  Fill field in new message from old message
            Add publish statement for this new message
```

2.6 Clang Tool

We are implementing the algorithm presented in the previous section with a tool which would perform the automated pruning of messages.

Clang is a C language family frontend for LLVM, and in this context is used as static analysis tool. ROS nodes (publishers and subscribers) can currently be written in Python or C++. The work in this section is a summary of how we are implementing the algorithm in Clang. It also details the progress we have made, with relevant results.

With Clang, we obtain an abstract syntax tree (AST) from the source C++ file of the ROS node. By implementing a recursive AST visitor in Clang, we can gather the relevant information from the given node.

The figure below details our approach for finding out the used fields in a subscriber.

For each Statement (S) Recursively Traversed in AST:

1. Visit Statement : S
    1. If (S is a BinaryOperator)
        1. Then, Visit Binary Operator (S)
    2. If (S is a Compound Stmt)
        1. Then, Visit Compound Stmt (S)
    3. If (S is a Call Expression)
        1. Then, Visit Call Expr (S)
    4. Similarly, we need to visit for IfStmt, ForStmt, DoWhile, SwitchStmt etc.
2. Visit Binary Operator:
    1. The approach remains the same as in CFG which is to getLHS and getRHS. Find for usage of a message field.
3. Visit Compound Stmt :
    1. For each part of compound Stmt individually process of message field presence. If present, mark as used.
4. Visit Call Expr
    1. Find all the arguments.
    2. Check for a message field usage.
        1. If found, then mark as used the message field.
5. Visit If Stmt, Visit For Stmt, Visit While Stmt etc.
    1. Extract the condition. Check for message field usage.

Figure 7: Subscriber analysis through a recursive AST Visitor in Clang

We have implemented the subscriber analysis which can detect the used fields given the source code of the subscriber node.

```
Used Message Field: score

Used Message Field: first_name

Sublisher Variable: sub

Callback Function Name:chatterCallback

Subscriber Topic: chatter_one
```

Figure 8: Output obtained from a sample subscriber analysis

Given these used message fields and the custom message file, we wrote an additional program to make a new lighter message (which only contains the used message fields)

```
MESSAGE PARSED
---------
uint8 : age
string : first_name
string : last_name
uint32 : score
```

```
FINAL MESSAGE (written to file)
---------
string : first_name
uint32 : score
```

Figure 9: Output obtained from a script which creates new ROS messages

3. Issues Related to Updating of ROS libraries

3.1 Problem Outline

As highlighted before, ROS is a meta operating system offered with a set of libraries which present an abstraction over the communication model which is at the heart of robotics systems. The set of libraries offer a wide variety of services ranging from communication to geometry and path planning. This set of libraries are usually maintained independently and may be updated before ROS is released in major updates.

Since a robotics project can be maintained and worked on by several developers, it is highly likely that some developers have an updated version of these libraries. Depending on the type and scale of change in the libraries, there might be functions which are used in the robot which now return unexpected values. This, in turn, can cause unexpected behaviour from the robotics systems.

3.2 Bug Reports

From a manual study of bug reports, we were able to compile a list which contains user- and developer-created reports of such instances wherein updating of libraries led to malfunctioning of a robotic system. The link to the complete study is given in the appendix entries[e] & [g].

Our main inference from these bug reports was that most significant changes are to be made to the build system, rather than the logic of publishers and subscribers. Therefore, previous research which applies to generic changes in libraries applies to this problem. We made research paper summaries of this previous work which can be found in the appendix entry[i].

4. Inconsistency Bugs

4.1 Problem Outline

Message fields are filled by a publisher node, and processed for information by a subscriber node. For any given field in a message, a publisher may fill it from a set of valid values. This set of values may not be consistent with what the subscriber is expecting.

A simplistic way to picture a subscriber would be in terms of a switch-case statement. The subscriber may check information from the published message and take appropriate action for it. This action may include publishing other messages, or directly controlling a connected component in the system.

This may cause unprecedented behaviour by the subscriber (or the robot in general):

- The subscriber may ignore a syntactically valid message. Unless explicitly checked for by the developer, no error/warning/information flags are raised if a subscriber does not take any action for a given message it receives. In certain cases, this might also lead to malfunctioning of the given robot.

- The subscriber may process an unexpected value from the message, which could lead to a crash depending on the way the subscriber handles the value.

- The subscriber may process an unexpected value from the message, and publish other messages based on the inferred information. This may lead to the overall malfunctioning of the robotic system.

4.2 Bug Reports

From a manual study of bug reports, we were able to compile a list which contains user- and developer-created reports of such instances wherein inconsistencies

between a publisher-subscriber pair led to malfunctioning of a robotic system. The link to the complete study is given in the appendix[b].

4.3 Symbolic Execution & KLEE

Symbolic execution is a technique in which arguments to a function are replaced by a symbol. This symbol is assigned different values during execution to determine which values make which parts of the program to run.

KLEE is a symbolic execution engine built on top of LLVM compiler infrastructure and operates on LLVM bytecode[5]. Even though KLEE is very useful, it has its limitations:

1. It cannot check all possible paths of a huge program. Even though it succeeded on many problems, it could not explore the simple `sort` utility in `coreutils` [6]

2. It is slow.

3. KLEE requires the program to be modified, which adds an extra overhead of creating scripts or performing these manually to use KLEE on a large project extensively.

4. One of the data type fields KLEE cannot handle is floating point data, which is quite common in robotics systems.

5. KLEE cannot handle objects of variable size.
    - Therefore, we cannot handle arrays whose size is not given
    - We cannot handle standard strings unless we pose a limit on its maximum length. The time increases sharply with an increase in the maximum size given.

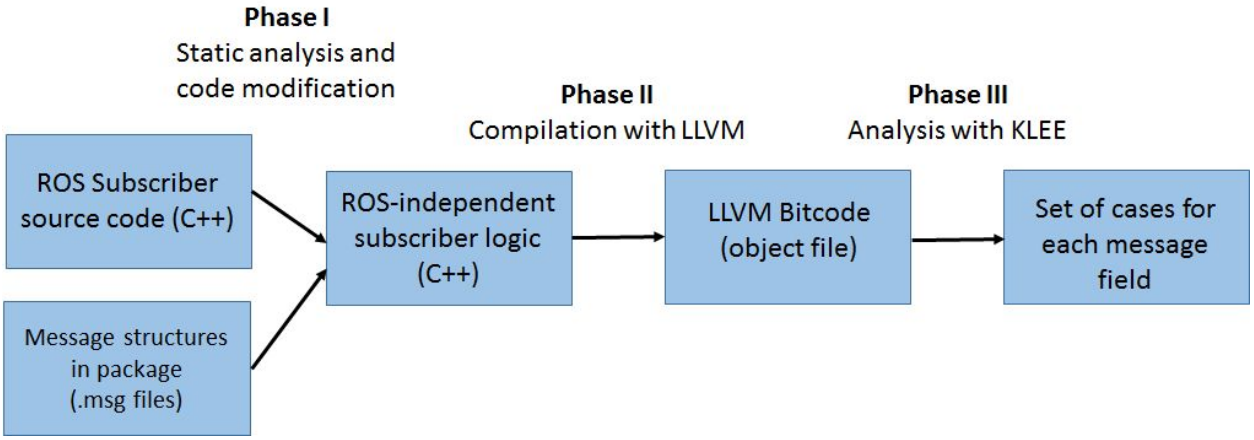
Despite its limitations, it has its strengths in creating test cases with a large coverage for the data-types it supports.

4.4 Solution workflow

We aim to solve these inconsistency issues by analysing the subscriber and identifying the values it is equipped to handle.

Each subscriber has a callback method which is executed whenever a message is received. By exploring all paths in this callback function for every message field, we can get a set of cases it is equipped to, and not equipped to handle. We can achieve this by marking all the fields of a message symbolic.

Since we're using KLEE as the symbolic execution engine, we have to generate the subscriber's LLVM bitcode first. Since ROS nodes run in an environment and KLEE cannot analyse ROS statements, we will first develop a version of the subscriber source code which is independent of ROS statements.



After we obtain the LLVM bitcode, KLEE can automatically run its analysis on the code and generate .ktest files which are extensive cases the source code can be tested with. The values which cause the program to crash are separately marked for convenience.

4.5 Running Example

To demonstrate the usability of KLEE in analysing subscribers, we have written an example, the source code for which can be found under the appendix entry [h].

5. Change in Message Logic

5.1 Problem Outline

All the components in the system have the ability to publish or subscribe to any channel by identifying them with their unique number or name. Thus, identifying components which depend on a specific component is not trivial. Even when these dependencies are limited to structured message exchanges, they are hard to grasp. These dependencies can be identified by observing the flow of messages in all components. Previous work towards solving this problem has included a static analysis approach which can identify dependencies and create a “message flow” graph out of the information.

A related problem is of identifying when there is a change in the logic of conditions of a message being published. Even a trivial change in such a condition could be the cause of widespread code modification in the complete system. Continuing the work started in *Extracting conditional component dependence for distributed robotic systems* by R. Purandare, J. Darsie, S. Elbaum, and M. Dwyer; we wanted to complete a tool which automatically iterated through all the changes in the given ROS code and identify where the logic of publishing a message was changed.

5.2 Development of tool

Previously, the tool was only working for if statements. The new modifications made to analyse other Clang Stmts as well. The new statements which were included are:

- `for` statement
- `while` statement
- `switch` statement

We had to call each stage of the individually in order to completely generate the output. Now a single script was made which does the following tasks:

- Call analysis for each stage one after the other without any external aid.

- Detects what is build type of the ROS package and then executes commands accordingly.

To completely automate the process we also wrote a script which iterates through all the commits in a github repository. We needed this script since we have to compare different versions for the same package. This script helped us get those different versions which we are able to compare and generate the results for.

6. Conclusions

Robotics frameworks have succeeded in providing great abstractions for developers of robot applications. And with the presence of such popular frameworks like ROS, we have been able to do an extensive study of robotics code, bug reports, and an understanding of the unique challenges faced by developers of robot applications.

Through our work, we presented some of these challenges and ways in which those challenges can be made easier through program analysis techniques. Using a variety of techniques like static program analysis and symbolic execution; and our knowledge of how these robot frameworks work, we have suggested ways in which the performance of the system and the pace of development in robotics can be increased.

7. Future Work

We will complete the tool to automate pruning of message fields and make this tool open-source and publicly available. Further, the technique presented in the report can be generalised to distributed systems and may be worked at as a protocol. If integrated into distributed system architecture, a plethora of current systems can experience a significant boost in network performance.

8. References

[1] ROS: http://www.ros.org/
[2] MOOS: http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php/Main/HomePage
[3] Understanding ROS Nodes: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
[4] rxgraph: http://wiki.ros.org/rxgraph
[5] KLEE: http://klee.github.io/
[6] Cristian Cadar, Daniel Dunbar, Dawson Engler: ***KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*** (OSDI 2008)
[7] R. Purandare, J. Darsie, S. Elbaum, M. Dwyer: ***Extracting conditional component dependence for distributed robotic systems*** (2012 IEEE/RSJ International Conference on Intelligent Robots and Systems)
[8] Garcia, Joshua, et al. ***Identifying message flow in distributed event-based systems*** Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013.

9. Appendix

[a] Message Pruning Results (1)
[b] Inconsistency-related issues - Bug reports
[c] Message Pruning Results (2)
[d] Five Publisher Message Pruning Results
[e] Updating of ROS libraries - Bug reports
[f] Message Pruning Results (Controlled Environment: 3)
[g] Updating of robots and dependent libraries - Bug Reports
[h] Integer message testing with KLEE
[i] Research paper summaries