Bridging ROS for Heterogeneous Integration in Mobile Robot Systems

Erik Jansson
ada09eja@student.lu.se

Tommy Olofsson
ada09tol@student.lu.se

Master's Thesis

August 14, 2014

LUND UNIVERSITY

Department of Computer Science

Supervisor	Klas Nilsson	klas.nilsson@cs.lth.se
Asst. Supervisor	Sven Gestegård Robertz	sven.robertz@cs.lth.se
Examiner	Mathias Haage	mathias.haage@cs.lth.se

Abstract

We investigate the difficulty of integrating disparate, heterogeneous systems which have not been designed to work together. Such difficulties may arise from differences in communication protocols or data formats, making an integration effort largely manual and labor intensive. The investigation is done in the context of integrating two different robot systems, one mobile platform running ROS (Robot Operating System) and one stationary two-armed ABB robot.

The thesis consists of two parts. First, existing solutions to this problem (or parts of it) are examined and evaluated for their applicability. After no suitable solution is found, a tool is then created which solves the problem of integrating non-ROS compatible devices with a ROS system. The presented tool is a program that generates modular bridging nodes between ROS and other systems.

Finally, the tool proves its value in the integration of two different robots, where one system also receives some additional changes for practical reasons.

Keywords: ROS, LabComm, System Integration, Heterogeneous Systems, ExtCtrl, VxWorks

Acknowledgements

The motivation behind our work comes primarily from the EU FP7-project PRACE [1]. We are thankful for the experience of interacting with the project members and for the greater context this gave our work. Special thanks to Alexander Bübeck and Benjamin Maidel at Fraunhofer IPA for letting us spend some time in their lab.

We want to thank the people at the computer science and control departments at LTH for all the help and guidance they have given us. A few specific persons are Klas Nilsson, for introducing us to the work they are doing and the constant mentoring, Sven Gestegård Robertz, for insightful discussions, especially about the `ros2lc_bridge` tool and for the comments on our report and writing style, Mathias Haage, who also gave us valuable critique on the presentation of our work, Anders Blomdell, for basically knowing everything about writing code, Anders Robertsson, for helping us in the lab with robots and Simulink, and lastly Maj Stenmark, who used our work and made us think further about the user experience. We also want to thank D. Acker for moral support during late coding sessions.

Finally we want to mention Erik Westrup and Oscar Olsson, who played a large role in the initial development of the Firefly library.

Contents

Chapter 1
Introduction

Problems can occur on many levels when one tries to connect different communication systems to each other. They can use different protocols, data formats, resource representations and, if applicable, different discovery and coordination mechanisms.

Our tasks exemplifies this well. ROS, introduced in Section 2.1, is a framework for building robot systems with clear and well defined mechanisms for communication between different parts. It is easy to write programs that use ROS, but only as long as one stays within the ROS ecosystem. Problems arise when systems cannot comply with ROS' requirements which puts a limit on which programs and therefore computers and devices can be included in a larger system based on ROS.

This thesis consists of two parts that deal with distinct but related problems. In the first part we investigate if it is possible to use some stand-alone method to communicate with a ROS system that is more flexible than ROS' mobel and makes it possible to integrate programs that cannot, or cannot *easily*, be modified to allow for inclusion into ROS. We do this by first looking at different frameworks that relate to network communication and distributed systems. Later follows a presentation of a tool that solves the problem.

The second part deals with a more tangible problem. A brief description, expanded in Section 2.4, is that a robot control system is to be moved from a work station to a mobile robot. The mobile robot is not capable of running the software for real time communication and manual configuration normally used at LTH. As a result, this software was modified to allow it to run on the mobile robot. Furthermore, the control system is to be integrated with ROS. We present modifications that solves these two problems as well.

Outline

Chapter 2 gives an overview of a set of relevant technologies. This chapter describes each piece of technology by itself, making no effort to relate them to each other. It lays the foundation for the understanding of the later parts of this thesis.

In Chapter 3 eight different technologies are compared and categorized according to four fundamental features. Due to time constraints, some of this work had little use during the thesis, but serves well to shine light on the problem domain.

Chapter 4 describes a large part of the work carried out during this thesis. It gives a conceptual overview of a *bridge generator* which aims to ease the integration of incompatible programs/systems with the ROS ecosystem.

The remaining part of the work is presented in Chapter 5. Here the changes made to a low level interface of an ABB robot controller to allow integration with ROS are described. Finally, the cost of entering a ROS system by use of the work described in the previous chapter is determined.

Chapter 6 describes what might be done in the future to improve our work, of which a short summary can be found in Chapter 7.

Chapter 2
Preliminaries

In this chapter, we present some tools and frameworks used throughout the thesis. ROS, a robotics framework, is presented in Section 2.1. The related technologies LabComm and Firefly, basically systems for stand-alone typed communication, are described in Sections 2.2 and 2.3. An extension to an ABB robot control system is described in Section 2.4.

Some other projects related to communications are presented at the end of the chapter. This highlights the problem space by mentioning what other solutions exist. Chapter 3 will continue this by briefly comparing and evaluate the features of each of these projects.

2.1 ROS

ROS is an open source framework which aims to encourage code reuse in robotics research and development by having a design that consists of loosely coupled components (from a source code point of view) [2]. The framework officially supports the Ubuntu Linux distribution while some other distributions and operating systems are listed as experimental or under development.

At runtime the system consists of a set of *nodes* which have their inter-node communication channels coordinated by a *master*. The master is a special node acting as a central lookup directory for nodes and their communication channels, of which there are two different kinds.

Services

Services operates according to a request/response model similar to RPC. When a client invokes a service it can choose to wait for the service to become available if it is not since calling a non-existent service produces an error. A service call is blocking and either fails or succeeds.

Services are often used to trigger actions such as spawning a character in a simulator or homing a wheel/motor controller in a robot. A service might also offer some kind

of computation.

Topics

Topics use a publish/subscribe model and usually carry the bulk of the data in a system (e.g., velocity measurements from motors or distance measurements from a laser scanner). In contrast to the client/service-model topics can be thought of as separate broadcast domains for messages with no notion of source or destination[1]. Instead every message published on a topic, from any node, is received by all subscribers.

There is no concept of a failed message transmission. In fact, it is common for newly established subscriptions to miss a few initial messages. Both publishers and subscribers may drop messages if their buffers are filled up. This might happen if a publisher writes to its outgoing buffer faster than the network can transmit, or if a subscriber fails to process incoming messages in time. This behavior might be suboptimal but must be kept in mind when creating applications. Topics should therefore not be considered for the implementation of stateful higher-level protocols to be used between nodes. Figure 2.1 illustrates the concept of publish/subscribe as it applies to a ROS topic.

The basic workings of the communication in ROS is shown in Figure 2.2 on page 12. Note that nodes are not constrained to the roles shown. They may very well be both a service provider and a topic subscriber, as evident in Chapter 4. An enumeration of the data and control flows in the figure follows.

1. A node requests a subscription to a `topic` and/or performs lookup of a `service` provider.
2. The master responds with information about publishers on the `topic` and/or the address of `service` provider.
3. A publisher registers its `topic`s with the master; and
4. receives a list of subscribers for each `topic`.
5. Messages on topics are then sent directly to subscribers.
6. Likewise, a service provider register its `service`s with the master.
7. A caller sends a request, which might include arguments, directly to the `service` provider.
8. The `service` provider executes the request and responds when done. The response might include return values.

[1] The publish/subscribe model mentions no specific nodes. At runtime, of course, they will be located and explicitly transmitted to.

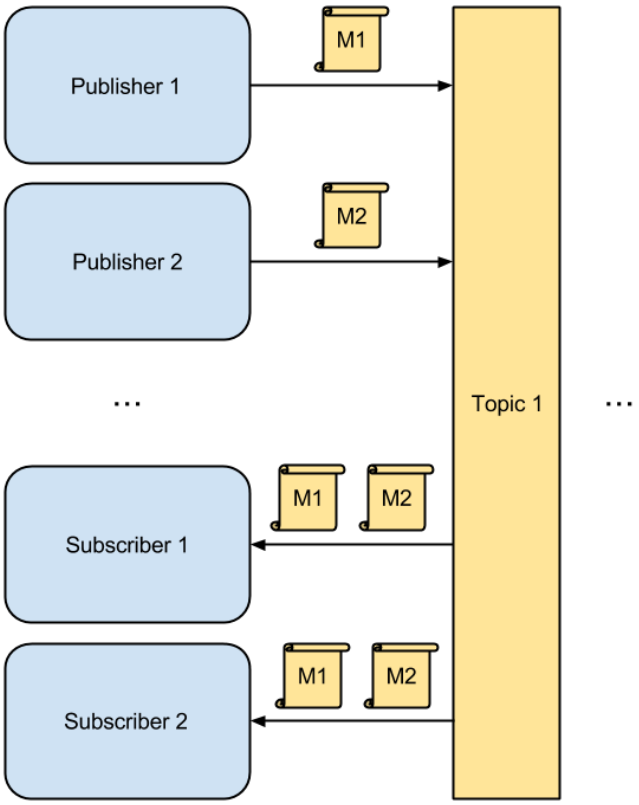

Figure 2.1: This illustration of a ROS-topic shows the broadcast of two messages. Note that there is no guarantee that messages sent from different publishers arrive in the same order at all subscribers, despite it being the case here.

All communication in ROS is typed. When the master has resolved a resource and two nodes are establishing a connection the hash of the type definitions are compared. This means that there will be no difference in definition of the type that is used. Strong conventions in the community regarding the use of standard types further assert the semantic meaning of any particular type.

The communication in a ROS system is done over TCP[2] and the nodes communicate with the master using an XML-RPC protocol. The wire serialization for communication between nodes is done according to a custom binary protocol that is not self-describing. There have been discussions about switching to some third party serialization framework, e.g., Google's Protocol Buffers, but no decision has been reached as of this writing [3]. The arguments for switching and thus breaking compatibility are to a large extent related to inclusion of smaller and more resource constrained systems.

[2]Some implementations have some support for UDP, but it is not widely used.

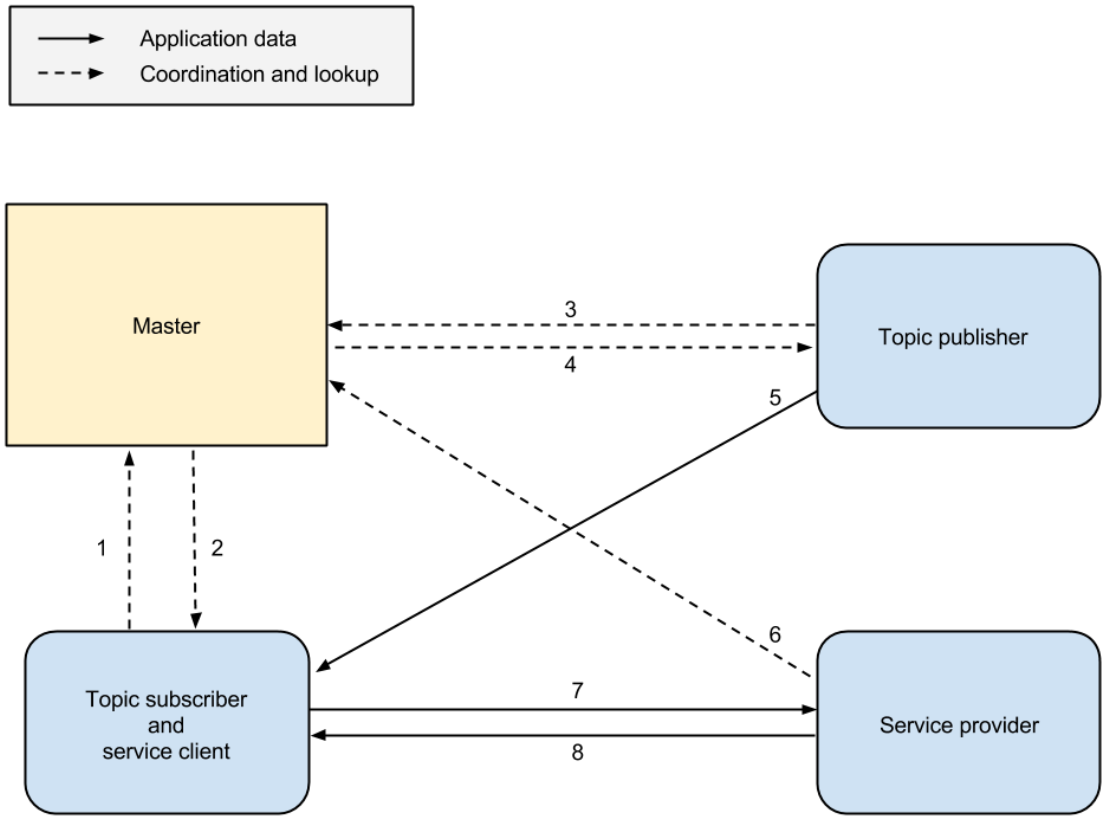

Figure 2.2: The master is responsible for lookup of services and of the publishers and subscribers on a topic while the flow of application data pass directly between the nodes.

2.2 LabComm

LabComm [4] is a serialization system developed at Lund University and used in the Robotics Laboratory for computer science and automatic control research. It has implementations in several languages and is used in a variety of different systems (e.g., desktop computers, ABB robot controllers, and embedded ARM processors for sensor data acquisition). LabComm is modular and one can configure how things such as memory and reading/writing on the transport level are handled. This fine-grained configurability combined with its binary format makes it easy to use with protocols suitable for real time networking.

In LabComm data is passed over channels and each channel has an encoder and a decoder at opposite endpoints. Data types are registered on encoders and decoders which allows those types to be serialized and deserialized on the corresponding channel. An encoder registration writes the signature of the type, and this has the effect of making LabComm *in-band self describing*.

The types are defined in `.lc`-files using a domain specific language similar to how types are defined in C. These are run through the LabComm compiler to generate encoding/decoding routines in the different languages that LabComm supports, currently C,

Python, Java and C#. Listing 2.1 shows an example of a simple `.lc`-file. The data types that can be encoded and decoded are called *samples*. Something not expressed in this listing is the ability to use arbitrarily nested structures which can reuse different previously defined user types. These are convenient features that are common in real-world `.lc`-files. A longer introduction can be found in [5, Appendices A and B].

Listing 2.1: Simple LabComm type definitions.

```
1   sample int an_int;
2   sample float a_float;
3   sample string a_string;
4   sample int fixed_array[3];
5   sample int variable_array[_];
6   sample int fixed_array_of_array[3][4];
7   sample int fixed_rectangular_array[3, 4];
8   sample int variable_array_of_array[_][_];
9   sample int variable_rectangular_array[_, _];
10  sample struct {
11    int an_int_field;
12    double a_double_field;
13  } a_struct;
14  typedef a_struct something_else;
```

2.3 Firefly

The Firefly library was created to make it easier to use LabComm to connect various programs. It is quite common to use LabComm to send a few well defined data types between two different programs over a TCP-stream. This is easy to set up and involves a limited amount of boilerplate code to set up the connection.

It is much harder to use LabComm in applications where UDP or Ethernet are better suited. LabComm requires the *signature*, a binary description of the layout of the data type, to be sent before any data of that type. Used over a protocol that does not guarantee ordering or even reception, these transmissions may be lost. Some protocol is required to make sure that the registrations are successfully transmitted between the two programs.

There also need to be some higher level protocol specifying what constitutes a connection. In the usual case of connecting two programs using LabComm over TCP, the connection is given by the underlying protocol. TCP is a stream-oriented protocol that takes care of the connection setup. Once the connection is accepted both endpoints are fixed. With a datagram-oriented protocol one loses this feature. There is no longer any functionality that takes care of port allocation and keeps track of connection state. One can simply send and transmit datagrams.

The amount of code that has to be duplicated in each application, especially in the server applications, to reimplement the lost functionality would be large. Furthermore, there are countless ways of implementing the functionality. Two programmers working on two different sets of applications would likely invent two different protocols.

Firefly was created to make the implementation of LabComm communication easy and its protocol well defined. It defines its own protocol for how connections are made and how types are registered reliably.

2.3.1 Flexible Network Model

The library defines three networking primitives which makes it possible to create an abstraction of different protocols. This limits the amount of code that would otherwise be duplicated between applications and makes it possible to use different protocols in a unified way. Figure 2.3 shows the relationships described below.

Link Layer Port (LLP)
The root of the hierarchy. Used to receive or open *connections* and can contain any number of them.

Connection
Represents a connection to another Firefly node. Contains any number of *channels*.

Channel
Contains one LabComm encoder/decoder-pair for two-way communication between Firefly nodes. Makes no restriction on the number of types which can be registered on the encoder and decoder. The type registration is done before transmission of application data, during the setup phase.

Internally, the library is implemented in an event-based manner. This means that after the user has created the LLP, an event queue and possibly a number of connections, most interaction with the library is done asynchronously via callbacks.

2.3.2 Modules

The design of the library from an implementation point of view revolves around the basic protocol, which is the largest of the four modules that makes up Firefly. It is implemented in a portable way and uses no system specific synchronization primitives or networking API:s. These dependencies are limited to three smaller parts, the aforementioned event queue, the transport layer and the resend queue.

The event queue schedules all the events that occur in runtime, for example the reception of a data sample or the close sequence of connection. Since the core protocol module cannot rely on platform specific synchronization primitives, this design was chosen to guarantee that no race-conditions ever arise when accessing internal structures.

The transport layer handles the specifics of sending a data sample over a certain implementation of some protocol. This abstraction means that not only is the API the same for different protocols, but it is the same for different operating systems and architectures. The only code that differs when using different layers is a few lines of initialization. This also makes it possible to use different transport layers simultaneously.

Lastly, the resend queue implements the resend feature of Firefly. It handles the queuing and timing of retransmissions of important messages if they are lost. This is primarily used for the handshake and type registrations but the user can also request that arbitrary data samples are resent.

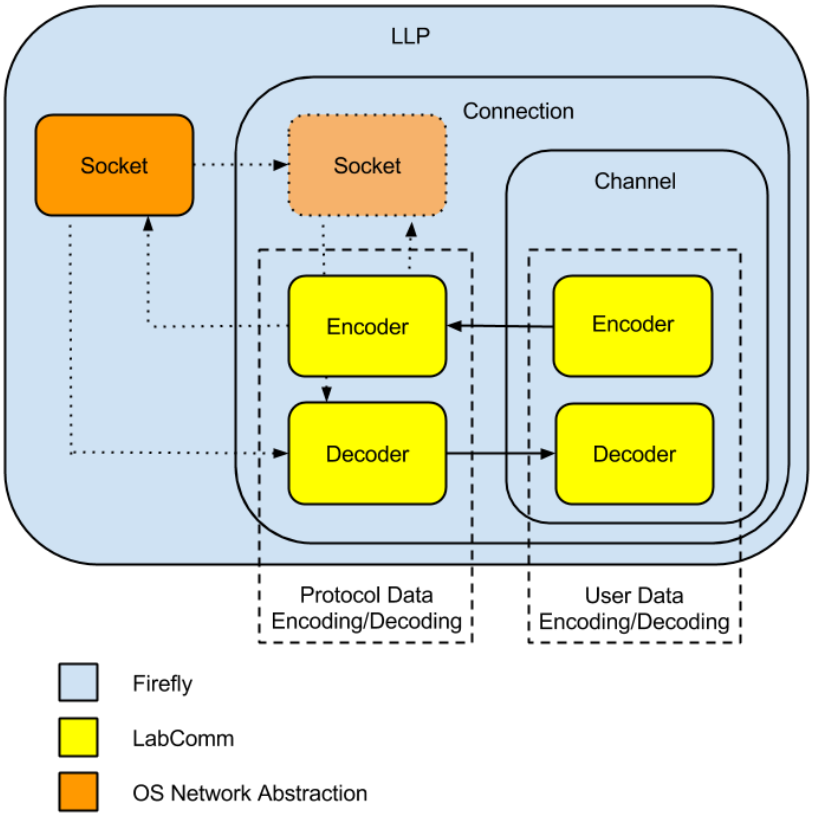

Figure 2.3: The network abstractions used in Firefly. The connection structure may or may not contain a socket, or similar handle, depending on whether or not the protocol used is connection based. If the protocol is connectionless this state is kept separately.

2.4 ExtCtrl

ExtCtrl [6, 7], short for *external control*, gives control researchers the ability to experiment with controllers for various ABB robots. The controllers are created using Simulink, a graphical tool in Matlab.

The way this works is by installing a program in the control loop executing on the ABB main computer that forwards this data to an external PC that executes the controller. New references are computed and sent back and reintroduced into the control loop. To achieve this in practice three pieces are needed: One program on the ABB computer, one containing the controller implementation, and one to coordinate the other two.

2.4.1 Robot Controller

The ABB control computer, which usually is located in the control cabinet, or in the case of the ABB Frida, inside the robot torso, is running the VxWorks real-time operating system from Wind River Systems. VxWorks consists of one huge namespace containing

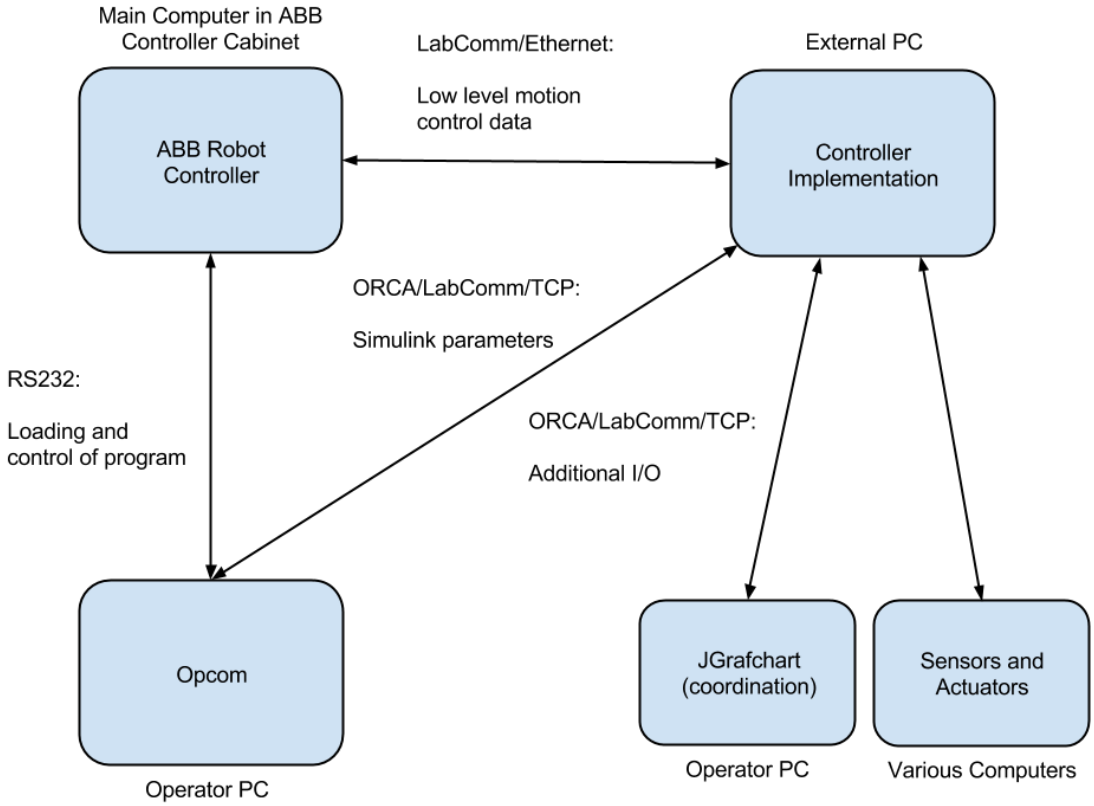

Figure 2.4: A high level overview of the ExtCtrl architecture with focus on the communication. Interactions between different parts of the ABB system are omitted.

all the loaded symbols. Symbols can be added and removed by invoking the dynamic linker from a UNIX-like shell that is provided on a 9600 baud serial port and typically used by connecting a PC running minicom[3], or something similar. The system features no memory protection and provides a debugger that is capable of printing and stepping instructions only, but not source code in the way one would be used to if from using any common debugger, for example GDB.

The main control loop (not the servo loops) runs on such a system at a frequency of 250 Hz. Callbacks can be installed to retrieve the current state of the motors (angles, velocities, torque, etc) and also to set the modified references. This interface is used by a program in turn used in the following way.

1. The program is loaded into the system.
2. A function is called from the shell and a task is spawned to wait for an incoming connection. The setup uses a custom protocol to ensure proper signature registration. Finally the ABB callbacks are installed.
3. Submission of values by the ABB system (observing) is started from the shell.

[3] `https://alioth.debian.org/projects/minicom`

4. Obtaining of values by the ABB system (control) is started from the shell.

5. Callbacks are removed.

Synchronously within these callbacks the data is encoded using an older version of LabComm (API and serialization) and sent over Ethernet to the PC specified as an argument during the state described in point 1 in the above list. The PC now has about 1 ms to respond before the emergency stop triggers and stops the robot.

2.4.2 External PC

At the other end of the connection the PC responsible for the actual computations resides. The specification thereof takes the form of a Simulink model, see Figure 2.5 for a partial example. From this model, C-code is generated which in turn is compiled and linked

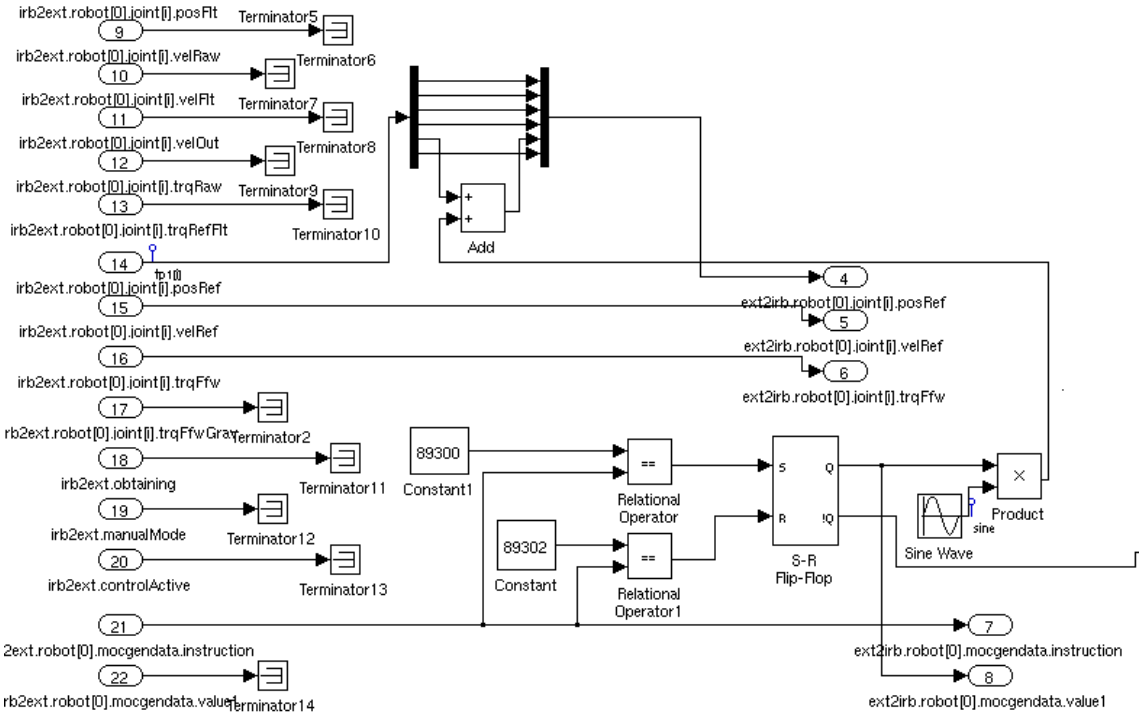

Figure 2.5: A part of an example controller implementation that waves one of the joints of the robot.

with additional generated code for the interface against the robot controller and additional I/O.

In addition to interfacing with the ABB hardware, it acts as an ORCA[4]-server. This enables the Simulink controller to use signals from, for example, extra sensors. To be able to respond to the messages from the ABB controller within the allowed time, the PC is running a Xenomai kernel and using the RTnet drivers for the network interface card. The actual controller program consists of three different pieces of code.

[4] A LabComm/TCP based coordination protocol for additional I/O.

1. One static file constitutes the client side of the connection to the ABB computer.

2. The actual controller code is generated by Simulink itself.

3. The glue code between the two is generated by a tool called rtw2 which is invoked by the Simulink build process. This also generates a LabComm-file containing definitions of some of the Simulink signals.

At runtime the controller is clocked from the incoming samples from the ABB Controller.

2.4.3 Opcom

Opcom is the user interface part of the two previously described programs. It loads the binary and changes state over the serial shell and also starts the compiled controller on the local system.

2.5 UPnP

Universal Plug and Play [8] has been used as a base for a number of SOA[5] frameworks. It builds upon a set of protocols, namely IP, TCP and UDP, HTTP, XML, and SOAP. The UPnP architecture consists of *devices* that provide functionality through various *services* and *control points* that controls the services.

2.5.1 Discovery

Discovery in UPnP is handled by the part of the UPnP protocol stack known as Simple Service Discovery Protocol (SSDP). This protocol is implemented by means of HTTP[6] over multicast UDP. A unit sends announcements via HTTP NOTIFY after joining and before leaving the network. The announcements contain basic information about the devices and the service types in the unit. The announcements also contain a URL to a document containing a detailed description about each device. The descriptions in turn are XML documents that, among much else, contain more in depth information about the services provided by the device.

This behavior by the devices themselves fully handles the task of informing present control points about their existence. In the case of a control point arriving, it can search the network for relevant services. Relevant services might mean all, those of a certain type, those on a certain device, or any specific service of particular interest. The search is initiated by HTTP 1.1 M-SEARCH containing the relevant search term. This prompts the relevant devices to respond using a message similar to the one sent upon their arrival, this time using unicast to the requester.

[5] Service Oriented Architecture

[6] The format is in fact not full HTTP, but uses part of the header field format defined in HTTP 1.1.

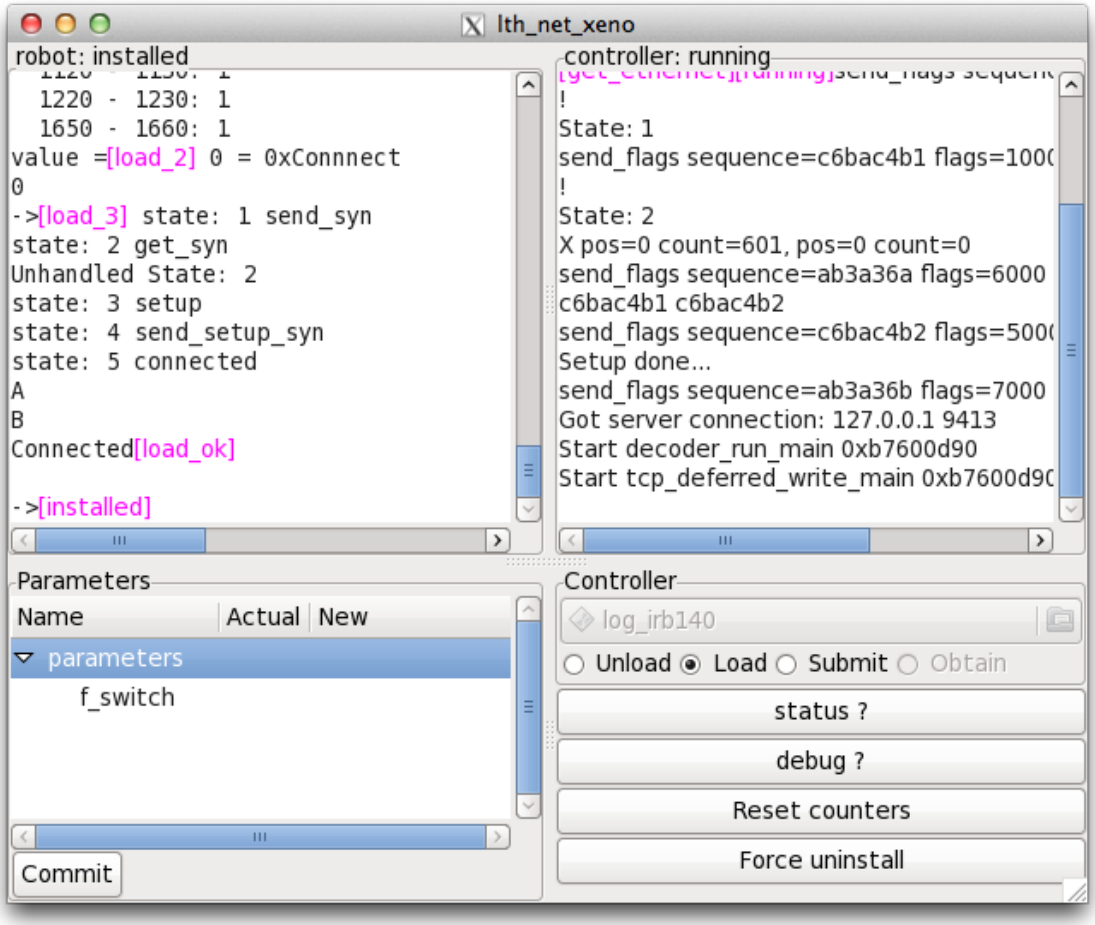

Figure 2.6: The Opcom operator GUI.
Top left: The VxWorks shell.
Top right: Controller output.
Bottom left: Parameters from Simulink model.
Bottom right: Button panel and entry for choosing (Simulink generated) Controller.

2.5.2 Service Interaction

The communication in UPnP takes place between the *control points* and the *services*, the latter one of which are located on devices. Devices might be nested on one physical hardware unit. It might also have multiple root-devices, however this is explicitly discouraged in [9].

The services present functionality in two ways. Control points can send *actions* to services. Some action is taken and if the service is specified to have a return value, it is returned to the invoking control point. An action has the effect of changing some state in the service. This state is presented to control points in the form of *state variables*. Each argument to an action corresponds to a state variable. Control points interested in the state of a service may subscribe to be notified upon changes in the value of these variables, if the state variable is defined to be *evented*. A state variable might also be defined as being

multicast evented. All changes to that variable will trigger a multicast notification. Note that this does not mean that there is single-variable granularity in the subscription model. A subscriber will be notified when *any* evented variable changes.

2.6 DLNA

DLNA [10] builds upon UPnP by defining standards for sharing media and controlling devices in a home network. The motivation behind DLNA is to define a least common denominator for a certain type of applications, i.e., streaming media protocols and control of multimedia devices.

2.7 Apache River/Jini

Apache River [11] (formerly Jini) was originally developed by Sun Microsystems (now Oracle Corporation) and builds upon the concept of *mobile code*. At the center of a Jini system is a lookup service to which services register themselves by placing a proxy-object in a lookup table. The proxy-object is defined by an interface[7] and knows how to communicate with the actual service.

A client in Jini queries the lookup service to find services that matches the interface it wants to use. The lookup service returns the proxy-object registered by the service matching that interface. Since the client knows the proxy-object's interface, it can now communicate with the service by dynamically loading the proxy-object and calling its methods. The proxy may use whatever means of communication it chooses to contact the actual service. The details of the communication may change at any time without changing the clients. They simply load the new proxy-objects to continue communication.

Jini also defines the concept of leasing; proxy-objects are leased for a limited time and once the time is up, a client must request the lease to be extended to keep using it. This helps expire connections to services should they unexpectedly go offline.

2.8 OBJE/Speakeasy

OBJE [12] (formerly Speakeasy) was developed at Palo Alto Research Center. It introduces the concept of *recombinant computing* and *serendipitous interoperability*. Like Apache River (described in Section 2.7) it relies on *mobile code* but the details and implementations of the systems differ somewhat. OBJE defines a set of minimal, generic interfaces that are required for devices (called *components* in OBJE) to implement.

When these interfaces are called on a service, they return custom objects which enables the caller to communicate with the service. Similarly to Apache River/Jini, these objects are leased for a period of time which can be extended. The different interfaces defined by OBJE are:

[7] The interface is also what the lookup table is conceptually indexed by.

Data Transfer
Defines a generic mechanism for transferring arbitrary data between components. A specific protocol is not defined since this would be too restrictive in some cases.

Aggregation
Allows for collecting several components into a group.

Contextual Metadata
Provides a user with contextual information about a component.

Control
Allows for presenting a control interface to a user.

2.9 Palcom

Palcom [13] is a framework for distributed computing developed in an EU FP6 project of the same name. It provides a middleware meant to separate the user applications and the functionality in a way that promotes reconfigurability at the end-user level. In Palcom *services* run on *devices* and answers to multiple *commands*. The communication takes place over UDP as XML, and the data type encoding is specified with a MIME type. Palcom also supports discovery of devices and services.

2.10 Protocol Buffers

Protocol Buffers [14] is a Google project and is in some sense the inspiration for Cap'n Proto, described below. The motivation behind the framework was to provide a flexible, binary alternative to less efficient transmission formats such as JSON or XML. The messages are defined in `.proto` files and encoding/decoding routines are generated by the Protocol Buffer compiler for C++, Java or Python.

When declaring a message type each field is given a tag number that is used to identify the field in the encoded binary format. Fields can also be either optional, required or repeated (zero of more times). The names are fairly self-explanatory. If optional members are missing during decoding, they are given a default value that can also be provided in the message definition. This means that, as long as new members are added as optional or repeated, messages sent from clients using old code can still be decoded by programs using the new message definition. The other way is also possible since unknown message members are ignored while decoding.

It is also possible to specify RPC service interfaces in `.proto` files that use the message types. The Protocol Buffers compiler then generates abstract interfaces that can then be implemented by the application to respond to the RPC calls.

2.11 Cap'n Proto

Cap'n Proto [15] is the work of the co-creator of Google's Protocol Buffers. According to the description a number of shortcomings have been corrected, as well as some function-

ality added. The improvements, which mostly lie in the implementation, include smaller binaries and optimized memory allocations. It also has more extensive RPC service definitions. As of the time of this writing the system has implementations in C++11 and Python, which limits the practical usability for the time being considering the popularity of the Java language and most important of all, C.

2.11.1 Serialization

The types defined by Cap'n Proto are similar to the fixed length primitive types in languages such as C and C++. Additionally there are constructs such as lists, structs, unions, and enums. There are also the types text and data.

The serialization format is designed to be used efficiently by most modern processors. This means fixed length primitives, as opposed to the otherwise popular *varint*, where integers are encoded in 7 bit per byte in a variable number of bytes to save space. To make up for this deficiency a packing scheme is defined for use cases where low bandwidth usage is of importance. This packs an entire message instead of individual members.

The serialization is generally similar to Protocol Buffers. This is especially evident in the design of the format, which is explicitly designed to be evolvable while still being compatible with programs using an old definition.

2.11.2 Remote Procedure Call

One big difference between Cap'n Proto and its predecessor Protocol Buffers is the way Cap'n Proto implements RPC interfaces. Similarly to Protocol Buffers, interfaces are defined in the `.capnp`-file and they may contain both methods and types. The interesting part is that when calling an RPC method, a *promise*[8], a placeholder for the result, is returned immediately. The major benefit of this comes when multiple requests need to be sent to one server in which case the promises can be batched and a single network request can be performed to get the result. Imagine for example an imaginary file system where we want to read a file `foo` in the directory `bar`:

```
1  bar = root.open("bar");
2  foo = bar.open("foo");
3  size = foo.size();
4  data = foo.read(0, size);
```

In the simple case, this would require four network round trips but Cap'n Proto's promise batching allows this to be done in a single network round trip.

[8]Promises are similar to *futures* found in other systems, but combining multiple requests is usually called *promise pipelining*.

Chapter 3

Comparison of Existing Software for Communication in Distributed Systems

A lot of work has been done to aid in the creation of distributed systems and systems consisting of heterogeneous components. Some projects have set out to solve a certain discrete problem, e.g. serialization, while other aim to be a complete framework for building a certain kind of applications. The second type of system might for example handle service description, service discovery, at least one communication layer, serialization, and even some kind of concurrency/threading abstractions.

A general trait among the larger frameworks is that they often evolve in a way that leads to them being hard to adapt to changes in requirements. Their scope might be very wide and provide a solution for many of the conceivable problem one might face, but at the same time make specific demands on the implementation. The application might for example have to be written in a certain language or it might depend on libraries available only to one platform.

The frameworks with smaller scope often solve a specific problem well. The main problem in this case is twofold. The frameworks are often written without the foresight of future extensibility, or a very limited and specific one. Furthermore, the possibilities of using two different frameworks as alternatives for a certain function in a larger one are often limited by inconsistent design and scope. Although this is not a surprising problem—rather something to be expected when comparing components from unrelated sources—available articles often overlook, or at least fail to mention, some of the complexities involved when discussing different solutions. Comparisons often feature frameworks which solve different problems, without mentioning of that fact. LabComm might for example be compared to Jini, even though Jini does not specify how services should serialize their data. Conversely, LabComm know nothing about services or discovery because serialization is the one and only function the system was developed for.

In light of these deficiencies we will analyze the different frameworks from a perspective of modularity. The section about each framework will discuss the following items:

A — Low level communication interface
Does the architecture define the low level communication protocol?

B — Serialization
Does the architecture define how data is serialized? If so, is it self describing? Is the description sent in-band?

C — Service interface
Does the architecture define how services/resources are represented to an application?

D — Service discovery
Does the architecture define how services are discovered in a networked system?

3.1 UPnP

UPnP covers points A and C of our model. The scope is well defined but limited by inexpressive types and a large amount of overhead.

3.2 DLNA

DLNA covers points A, B, and C, with the exception that the serialization (B) supports no user defined types and the service interface is predefined and static. While this might be suitable for its particular use case, it is to limited for the task at hand. Relative some other examples in this chapter, not much inspiration are to be found here.

3.3 Apache River/Jini

The system is realized by relying on Java to provide a homogeneous environment and to abstract away the lower levels of devices. One significant drawback of this is that each device is required to somehow run a JVM. There may also be security considerations of loading arbitrary code into devices from multiple manufacturers. The system also only covers point C; both A and B are left for the implementer of the proxy-objects and the service. This increases flexibility but also the workload when adding new devices and services.

3.4 OBJE/Speakeasy

As with Apache River, neither A nor B are covered by OBJE. Such details are left to the implementations of the custom objects. It does define how to use services which covers point C. It also does not provide service discovery.

3.5 Palcom

Palcom handles points A, C and D and bears a slight resemblance to UPnP with regard to the service model.

3.6 Protocol Buffers

Protocol buffers are similar to LabComm but with some major differences. They are not self describing, nor can a channel be multiplexed. An interesting feature that LabComm does not have is the support for inconsistent versions of type where a member is added or removed at one side. They cover point B and C but the serialization format is not self describing.

3.7 Cap'n Proto

Being heavily inspired by protocol buffers, they are of course the most similar software. They share the support for adding members, but Cap'n Proto also supports a feature it calls *packing*, which is simply compression of messages. It also have extensive support for RPC. As for protocol buffers, points B and C are covered and the serialization format is still not self describing.

3.8 LabComm

Unlike protocol buffers and Cap'n Proto, LabComm sends the message description in the communication channel once before sending data. This has the effect of making the serialization format *in-band self-describing*. This means that it is possible to decode messages of previously unknown types. Of course, this does not solve the problem of understanding the decoded types which still requires human intervention or some kind of semantic machine interpretation, which is work in progress at this time.

Another difference is LabComm's concept of communication channels (see Section 2.2 on page 12). Once established, these can be multiplexed in the sense that a number of data types can be sent over a single channel. This stands in contrast to Protocol Buffers, where an instance of a certain type is initialized using a stream.

LabComm does not specify any way of declaring services or how to perform discovery of these. Its purpose is to cover point B in the established categorization. It is currently being used successfully when building frameworks with greater scope to connect a wide variety of devices.

Chapter 4

Bridging ROS with incompatible systems

This chapter examines some problems caused by ROS' homogeneity (Section 4.1). It also presents an existing solution that tries to address this along with its shortcomings (Section 4.2). After this, we propose a new solution that addresses these shortcomings while still solving the original problem (Section 4.3). Lastly we present a couple of example use cases for the bridge that have been instrumental in driving the design and implementation (Section 4.4).

4.1 Problem Description

ROS places specific requirements and restrictions on the architecture of the individual systems that make up the larger, distributed ROS system. For example, the only officially supported platform is Ubuntu, a Linux distribution for the x86 architecture. This specificity is mirrored in the design of the communications library, which is very much integrated into the rest of the system, and cannot easily be used stand-alone.

This makes it hard to add or move parts of the system to other devices with different designs. A possible scenario where this would pose a problem is one where part of the control of a robot is to be moved to an external device where the requirements of ROS cannot be satisfied. This might be because the second system has other requirements that take precedence and prevents us from converting it to run on an appropriate platform.

The problem of ROS being loosely coupled only at compile time and fairly static at runtime, because of its whole-system compilation model, can be mitigated in different ways. A common one is to write bridge nodes for different purposes. This may be an appropriate solution for one-offs, but will invariably lead to duplication of work whenever more devices are to be integrated.

4.2 Existing Solution

The issue is recognized by the ROS community and one proposed solution consists of the `rosbridge_suite` package [16, 17]. This software defines a protocol that can be used to perform most of the actions that can be taken by an ordinary ROS node, i.e., topic subscription and publication, service calls etc. The package also includes a server that runs in the ROS-system and the corresponding client library that can be used by various devices. The documentation states the following:

> "The specification is programming language and transport agnostic. The idea is that any language or transport that can send JSON can talk the rosbridge protocol and interact with ROS."

Additionally, the independent website for the rosbridge project [16] answers the question "What language can I write a rosbridge client in?" in the following way.

> "Rosbridge clients can be written in any language that supports WebSocket. Many rosbridge examples are for JavaScript and web clients, but rosbridge is in no way restricted to just the web browser. For example, open source WebSocket client libraries are available for python, java, android, C# / .NET, C++ / Boost and many more."

There are a couple of problems with this that mostly strike against resource constrained devices, but also include some fundamental design issues. The following sections discuss these problems in-depth.

4.2.1 Transport Protocol

WebSocket [18] is a protocol that was created for the sole reason of providing browser based applications with a lower level alternative to XMLHttpRequest, which has high overhead, something more like a classic BSD socket. This was implemented using a HTTP handshake and a TCP-based protocol.

Because of this background, WebSocket is constrained to only work on top of TCP which might not be feasible for all applications. Some devices need to transmit data at such high rates, but with low enough computational overhead, that an entire TCP/IP stack is far too large. In addition, TCP is designed to reliably transmit data over vast distances. If the communicating parties are connected to the same network switch, the probability of packet loss is reduced substantially and the features of TCP becomes less appealing. In some applications, e.g., streaming of video or data from a laser scanner, one does not care about losing a few data packets at all. All these reasons may conspire to discourage the use of a TCP-based protocol such as WebSocket.

4.2.2 Data Format

The choice of JSON as a data and serialization format is inefficient both from a computational and encoding size perspective. It requires the data to be sent over the wire as

text with the names of the members included in every message. The rosbridge protocol adds another level of this by specifying that the operation, e.g., `publish`, is encoded as a non-abbreviated string.

A comparison between JSON and a common method for encoding integers is shown in figure 4.1. The situation is a bit more complex for floats as the standard does not specify any maximum length [19][1].

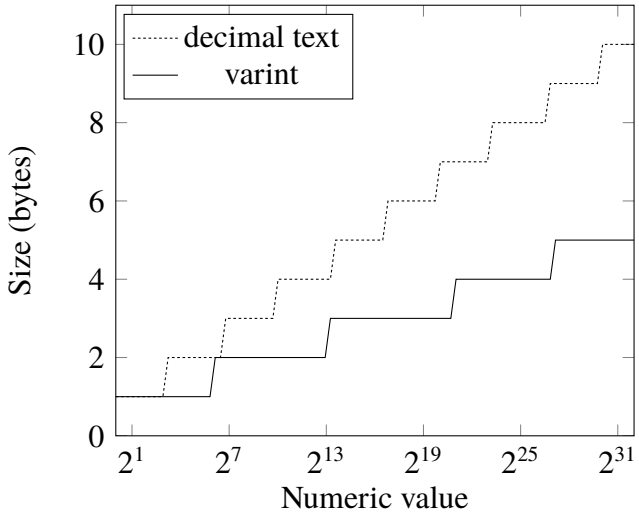

Figure 4.1: The number of bytes required to encode a 32-bit unsigned integer as JSON compared to 7-bit varint. The figure only shows the space required for the actual value, but JSON would usually need to send a name/key with every value as well as separator characters making is significantly less efficient.

An additional caveat is the conversion from plain text/JSON to native data types in strictly typed languages. Equivalent data structures and conversion routines will have to be manually authored for the types used. Those would in all likelihood use some third party library for parsing JSON, and custom functions for populating the native structures. With the availability of proper serialization frameworks like those described in Section 3 on page 23, this leaves much room for improvement.

4.2.3 Integration Effort Required

It is important to note that external nodes have to be written to conform to the interface specified by the bridge. The external program must act as the client and use the publish/subscribe protocol specified by the bridge. Furthermore, there is no flexibility to reshape messages with different layouts, no way to combine or split messages that happens to be combined in one of the systems and no way to throttle a topic. There simply is no support for any advanced features that would ease the integration with some existing system.

[1] In practice there is of course some upper limit on how many digits makes sense to transmit before scientific notation is used. For example, the `json` module used in Python seem to limit the length of an encoded number to 22 characters.

4.2.4 Summary

The problems mentioned in the previous sections mean that the integration of independent systems with a ROS system still has some unsolved complications. We believe that a more flexible design is essential for enabling ROS to work together with other systems in a general way. This includes a set of features not provided by the rosbridge package which would allow integration even when the connection model is incompatible. This extends to the types used in the two systems. A device should not be forced to conform to types used in ROS since this would force the device to be rewritten. A set of related topics in ROS might for example be sourced from an external device as a single data type. These requirements implies that there should be some way to specify the mapping of incompatible data models in the bridge node. In order to include devices of incompatible connection models, the bridge should also have to provide the ability to specify that an external device expects to act as a server rather than connect to the bridge.

The bridge should also use a more efficient way of serializing data. The problem of incompatible encoding formats is basically inescapable. With encoding methods that have stricter types (such as Protocol Buffers), both parties need to know how to decode the same types to communicate. This problem can be alleviated by using an encoding method that is self-describing. This would allow types to be decoded without explicit decoding routines being present.

With this in mind, we will present our solution which solves these problems in the next section.

4.3 A LabComm based bridge generator

Instead of implementing a general bridge (as rosbridge_suite does), we aim to create a generator which creates more specific bridge nodes. This has a couple of advantages, including amongst others:

- The bridge can be simpler since it does not need to keep track of all topics that exist in the ROS system but can be limited to those specified when running the generator.
- It allows “segmenting” topics and services and to be selective about which ones to export from the ROS system.
- Multiple bridges can be generated to handle different collections of topics. In this way subsystems can be handled independently, possibly exposing different interfaces.
- The generation can be extended to include any number of special features that change the data and how it is handled, all to better suite the case at hand.

4.3.1 Bridge Generation

The bridge generator can be run as a standalone program, as well as be invoked as a ROS service. In the service case, the service receives a configuration when called and generates

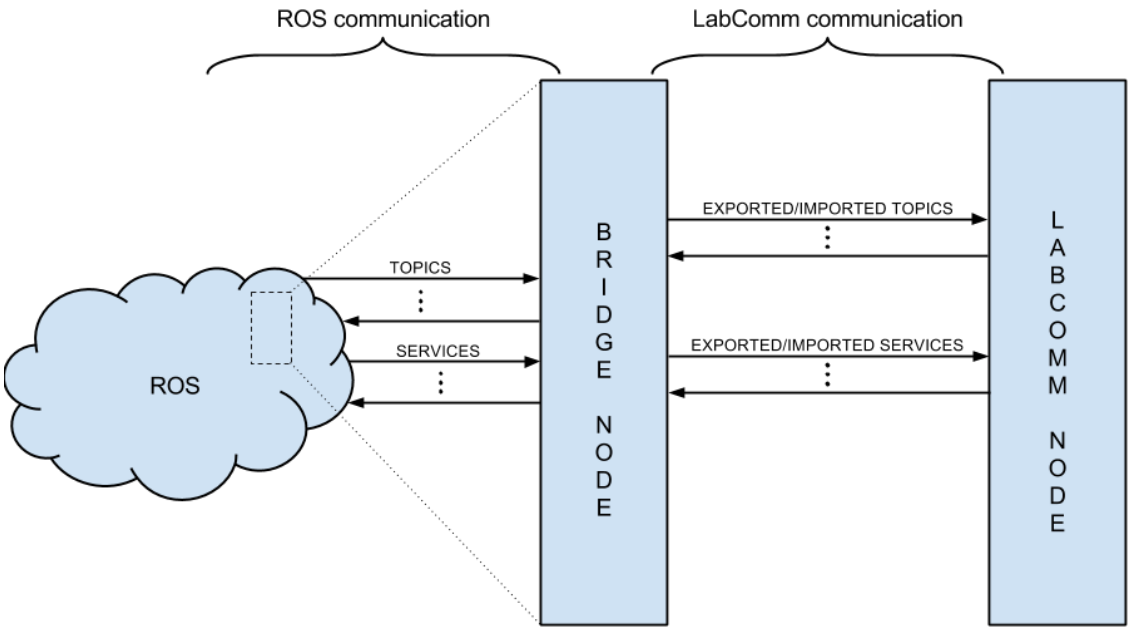

Figure 4.2: Conceptual design of the bridge.

a new bridge based on that configuration. When the generator is run, it inspects the ROS system's state and, based on an XML-configuration file, generates a new ROS package with a single executable, the bridge node. A very simple bridge configuration is shown in Listing 4.1. More advanced configurations can be found among the examples in the project repository[2].

Listing 4.1: A minimal bridge configuration.

```
1 <bridge name="pingpong_bridge" port="7357">
2   <imports>
3     <topic name="/ping" />
4   </imports>
5
6   <exports>
7     <topic name="/pong" />
8   </exports>
9 </bridge>
```

The bridge specifies a protocol (similar to that of rosbridge_suite) which mirrors ROS' publish/subscribe protocol. This allows external devices to be written like ROS nodes but with more lightweight and efficient communication and serialization methods (more on that later). When the generated bridge is run, it will perform conversion of data flowing through it. Conceptually, the bridge lies on the border between the ROS system and the rest of the world. Practically, it is a ROS node and thus resides inside the ROS system. It follows from this that the bridge has the same requirements and capabilities as an ordinary ROS node. It is responsible for receiving external devices' topic subscriptions and publications while also letting them invoke ROS services asynchronously. This differs from the service model in ROS in which services are synchronous.

[2] http://git.cs.lth.se/robotlab/ros2lc_bridge/tree/master/examples

The generator that has been developed during the thesis is written in Python and can generate bridges implemented in Python as well as in C++.

The following sections describe how this new implementation addresses the shortcomings of the previous bridge.

4.3.2 Transport Protocol

To be more flexible regarding which transport protocol is supported, we opted to use the Firefly library (see Section 2.3 on page 13) for the C++ bridges. This allows the bridge to support lightweight communication protocols and easily switch between protocols in the future.

Another feature that Firefly enables is that the bridge can easily use multiple transport protocols simultaneously (e.g., UDP and raw Ethernet at the same time). This would allow devices that use different protocols to be integrated into the ROS system. However, due to time constraints, the feature has not been implemented in this version of the bridge.

4.3.3 Data Format

LabComm is used instead of JSON. It provides fast and efficient serialization of data and minimizes the bandwidth required to send it, as well as the processing time required for serialization. As mentioned previously, the fact that LabComm data streams are self describing means that the bridge would be able to decode data which was previously unknown to it. This does mot mean that the type is semantically understood, but it allows a future extension where the user can be presented with what *is* known. She can then use this information to connect the components in some high level fashion, for example, by giving the bridge or the generator a user friendly GUI. This might be an alternative way of specifying the bridge configuration.

4.3.4 Integration Effort

Two configurations are possible for an external program. The most flexible alternative is to have it send subscribe and publish requests, according to the bridge's protocol, for each topic it intends to interact with. This has the advantage of being familiar to people who are used to develop for ROS. It does mean, however, that the program needs to be written with knowledge of the bridge and the model it uses. The alternative is to consider the external program as a server that expects the connection to originate from the other party. This is specified as a configuration option in the configuration file used by the generator and will cause the bridge to initiate the connection to the external program at launch. Programs that already use LabComm can be connected to a ROS system without any changes. If the program does not use LabComm, only the serialization method needs to be changed since the bridge can adapt to different connection models.

Another hurdle to seamless integration of existing devices is inconsistent types. ROS has a strong convention of using a set of standard types but such conventions disappear when you go outside the ROS system. Thus, types may not map exactly to each other. We solve this problem by enabling custom conversion functions to be specified. These

functions can receive multiple inputs as well as multiple outputs, i.e., they support a fan-in/fan-out pattern of types. This method is very flexible and allows any number of types to be converted to any other number of types.

Since the new data types must be "addressable", the bridge handles custom conversion by letting the user create *pseudotopics*. These pseudotopics can be used by external clients as if they were a proper topic with the desired structure. This structure is described by specifying the name of the pseudotopic, a file containing the conversion functions and a LabComm type in the configuration file.

Since data on topics can arrive in any order and at any time, when aggregating multiple data sources into one (or a few), the problem of deciding when to send the aggregated type arises. We solve this by providing a set of different *trigger policies*.

Single update
Send all outputs whenever one input is updated.

Full update
Send all outputs only when all inputs have received an updated value.

Periodic sampling
Send all outputs with a predetermined interval regardless of the inputs.

Custom
The user specifies a custom rule to determine whether or not an update should trigger a transmission. In this case, another function is provided by the user (in addition to the one performing the conversion).

4.3.5 The Generator

As mentioned earlier, the purpose of the generator is to gather information about the topics used in and the services provided by the system. The various ROS message definitions will be converted to equivalent[3] LabComm type definitions. Messages on different topics will be sent as different LabComm samples. Service definitions will be converted to two different samples, one for the parameters and one for the return values. Like ROS messages, the samples can have nested types.

The generator takes an XML-formatted configuration file as input. This determines the following configurables:

- Which topics and services the bridge should interact with,
- which port to listen to,
- which (if any) external devices to connect to at launch, and
- conversion functions for converting between incompatible types.

[3] Some limitations apply, see Section 4.3.6

The collected information is used to create a bridge capable of translating messages to LabComm format and vice versa. The generator is modular and pluggable so it can generate bridge nodes in several different programming languages. The two currently implemented are Python and C++ since those are the two supported by the ROS standard install.

Python Bridge Specifics

The Python implementation relies heavily on Python's dynamic nature to setup and dynamically configure itself to the specified configuration. This includes setting up data types and objects to communicate with the ROS-system. When a connection is received by the bridge, a new thread is dispatched to handle it without blocking the bridge itself. If the configuration specifies any external devices to connect to, the bridge starts threads to handle these at start-up as well. Likewise, when a request to a service arrives from an external device, a worker thread is started to handle the call.

C++ Bridge Specifics

The C++ bridge is very similar to the Python one with some minor differences. Topic subscriptions and publications work similarly to the Python bridge. However, service calls do not require spawning a new thread since the `roscpp` library handles this automatically.

4.3.6 Discussion

There are a couple of drawbacks with the new implementation. One example is that, since Firefly does not exist in a Python version, the Python version of the bridge is using LabComm over regular TCP. This is a bit inconsistent but has its advantages as well. We have found several use-cases in which an existing device was to be integrated into a ROS system and was written using a legacy TCP-LabComm library which was nontrivial to replace. Using the Python bridge allowed effortless integration of the device into ROS.

Another missing feature of the bridge is that it does not handle discovery of components outside ROS. Whichever side is configured as a client must connect to an explicit address. It is easy to imagine another layer which relies on some sort of discovery mechanism (perhaps a central repository as in ROS) to provide this.

A final issue worth keeping in mind is the difference in primitive types available in ROS and LabComm. ROS has a larger number of them. The `time` and `duration` types are generated as user types, but worse is the fact that LabComm does not support unsigned integers. The automatic generation of LabComm types selects the next larger *signed* integer when an unsigned is used. This allows all data from ROS to be represented losslessly, but the other direction might lead to trouble if the bridge is used carelessly. The widest unsigned type has no wider signed type to use, and might theoretically be a problem. An exerpt of this code is shown in Listing 4.2.

Listing 4.2: Part of the automatic conversion of ROS types to LabComm types. Lines 1-18 show the mapping ROS-LabComm and line 22-25 show the handling of the time-related types.

```
ros_prim = {
  'byte'     : 'byte',     # Deprecated alias for int8.
  'char'     : 'short',    # Deprecated alias for uint8.
                           # Bad for incoming messages.
  'bool'     : 'boolean',
  'int8'     : 'byte',
  'uint8'    : 'short',    # Bad for incoming messages.
  'int16'    : 'short',
  'uint16'   : 'int',      # Bad for incoming messages.
  'int32'    : 'int',
  'uint32'   : 'long',     # Bad for incoming messages.
  'int64'    : 'long',
  'uint64'   : 'long',     # Bad for outgoing messages.
  'float32'  : 'float',
  'float64'  : 'double',
  'string'   : 'string',
  'time'     : 'time',     # Handled as typedef.
  'duration' : 'duration' # Handled as typedef.
}
  # ...

  labcomm_file.write('''
typedef struct { int secs; int nsecs; } time;
typedef struct { int secs; int nsecs; } duration;
''')
```

4.4 Example Use Cases

This section presents a couple of use cases we used to drive the development of the bridge generator. It illustrates, in a tangible way, the problems with the rossuite_bridge and how we solved them. While some of the examples by themselves may be of limited practical use, they proved invaluable during development to affirm and reject proposed solutions. They also served as practical test cases for the software.

4.4.1 Force/Torque Sensor

The first example is one which exists in the robot lab at LTH. A program reads data from a force/torque sensor and when clients connect it sends that data to the clients over a network. The hardware (i.e., the actual force/torque sensor) is connected to a small embedded system which runs a real-time operating system. Thus, it cannot be converted to a ROS node without hardware upgrade and/or rewriting the program. The actual protocol is very simplistic; once a client connects, a stream of the sensor's values is sent to the client until it disconnects. This scenario is illustrated in Figure 4.3.

This example indicates that the bridge should be more flexible about its connection model rather than just extend the reach of the ROS interface the way rosbridge_suite does. However, it should not restrict the bridge from also supporting the ROS concept since it is very flexible for nodes to be able to subscribe and publish at their leisure during runtime.

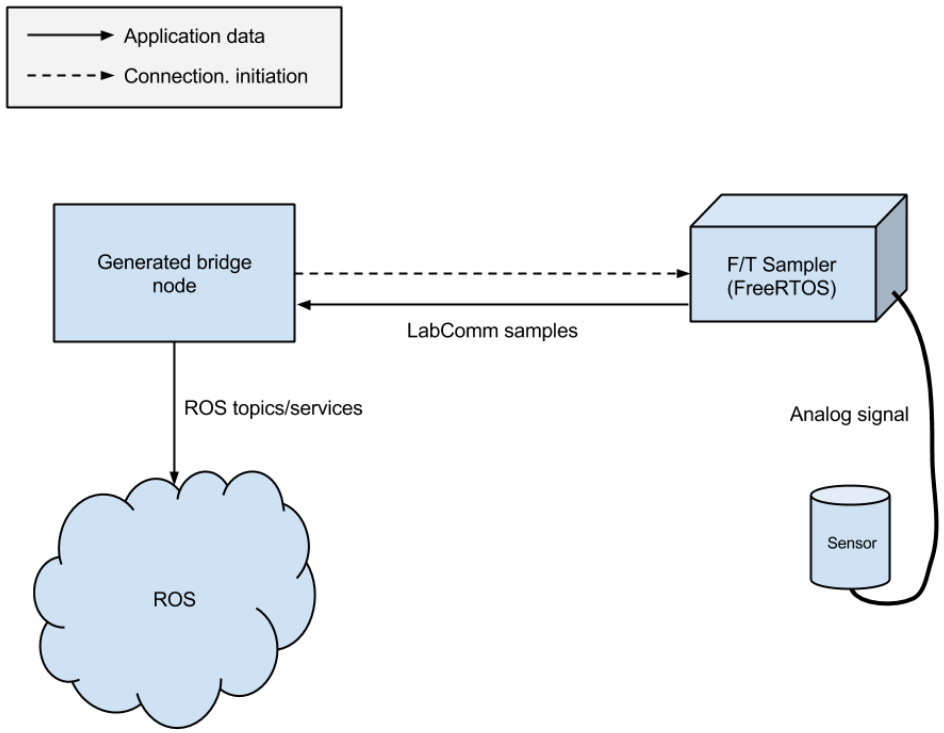

Figure 4.3: An example of a sensor used in the LTH robot lab. An embedded system samples force and torque and these are imported into a ROS system.

4.4.2 The Trajectory Generator

This example is larger, more elaborate and consists of two main components. The first is a trajectory generator inside ROS which uses some algorithm to compute four velocity set-points from three-dimensional positions in space. Each value (e.g., the position value of one of the axes) is sent on an individual topic (see Figure 4.4).

On the other side, outside of ROS, there is a manipulator which provides a control interface based on two different LabComm samples. These types correspond to the topics but are composite types (also seen in Figure 4.4). The two representations are thus designed to serve the same purpose, but they are structured differently.

This scenario indicated the fact that the bridge needs to provide a plug-in functionality for custom conversion between types. Worth pointing out is that this scenario could also use the same statically defined connections as the force/torque scenario. These features are orthogonal and may therefore be used independently.

4.5 Comments

The so called “custom conversions” were first planned to be specified using a custom syntax. This idea was later scrapped when we realized that this would severely limit the flexibility of the feature and make it impossible to specify truly *custom* conversions. An

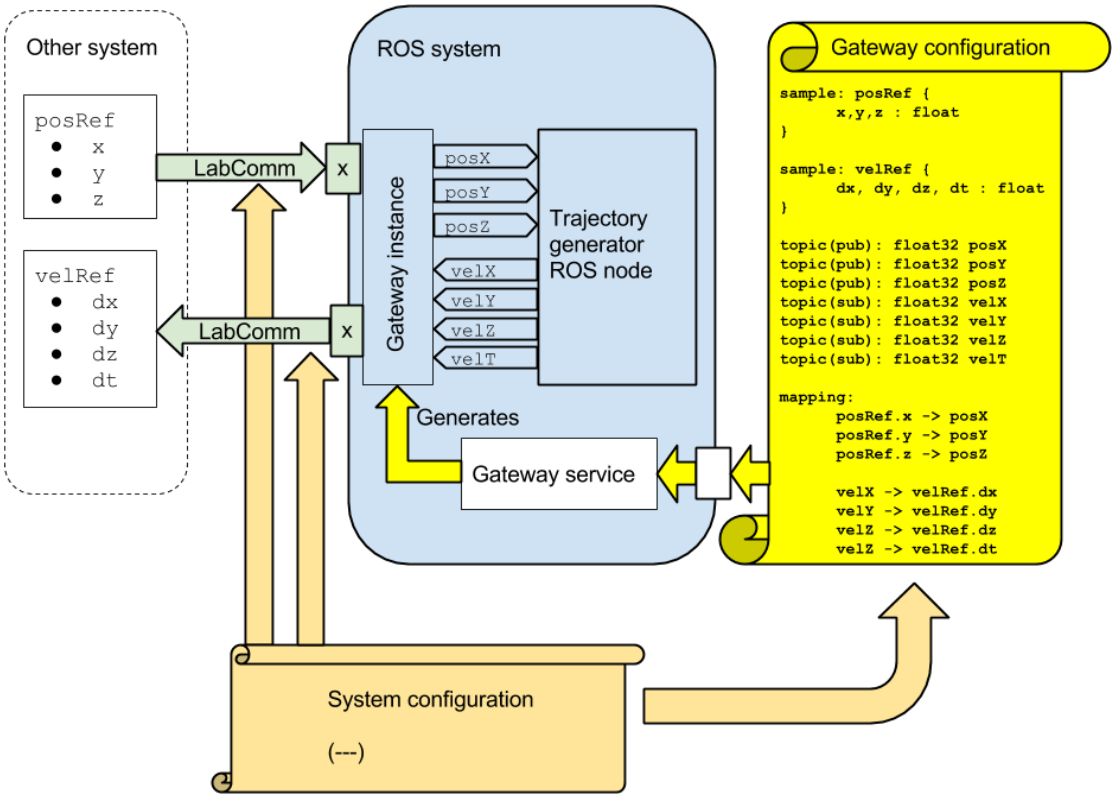

Figure 4.4: The *trajectory generator*-scenario is one of the most elaborate examples for which a bridge can be generated. The figure also shows how the bridge generator could be invoked as a ROS-service, possibly through its own generated bridge. This makes it possible to examine and tap into a ROS system during runtime.

example of such a language can be seen in the lower right part of Figure 4.4.

A domain specific language would make a complex conversion impossible if it was not made sufficiently advanced, in which case one is basically implementing a general purpose programming language. For this reason we chose to specify the conversions in the language to be generated. In the simple case of just mapping different names the proposed language looks rather like assignments in the target language anyway.

This has the down side of requiring different implementations of a conversion for the two languages supported. However, we do not consider this an issue because of the different communication protocols used by the different implementations. One would know from the beginning which implementations are needed. If the communication is to take place over LabComm/TCP, one need to use the Python back end. If Firefly/UDP should be used, then the C++ back end is needed.

There were plans to use Lua [20] as a more user friendly way of specifying conversions when using C++. However, due to time constraints, this idea was abandoned but it is an interesting future extension.

Another possible extension to the C++ implementation that have not been sufficiently

examined is to use the feature of Firefly which makes it possible to easily change the underlying protocol. An additional tag in the configuration file could be used to select either TCP, UDP, or plain Ethernet. All these protocols are already supported by Firefly and one could argue that different devices should use the protocol that best fit the nature (size- and time-wise) of the data transmitted.

In fact, there is nothing preventing a bridge from using any combination of protocols simultaneously. This could simplify integrating existing devices using different protocols with the same bridge.

Chapter 5
Modified ExtCtrl

The ExtCtrl architecture described in Section 2.4 has been adapted for its integration with a mobile platform. The robot to be used on the mobile platform is the ABB dual-arm concept robot (Frida). It has two arms, each with 7 joints.

The motion control is normally performed entirely by on board computers, which leaves the user controlling the arms in a point-to-point manner via ABB's RAPID language. This has proved to be insufficient for some tasks. An example of this is the handling of pegs whilst mounted on a mobile base, work currently carried out at as a joint project between Lund University Fraunhofer IPA in Stuttgart, among others.

This protocol described in Section 2.4 was not designed with handover and reconnection in mind. It is mainly focused on providing a researcher/student with a way to run experiments. As the end goal of the PRACE project is to have a demonstrator installed at an industrial assembly site, with the possibility to leave the ABB robot at a station for independent tasks, some changes are in order.

5.1 Changes

First of all the behavior of the server program running on the ABB IRC5 robot controller was changed in such a way that it does not require coordination via the shell and the serial port. The new behavior is for the program to install itself into the control loop after a connection has been successfully established and prepared. This was done by using the Firefly library, introduced in Section 2.3. It replaces the custom protocol previously used during setup, and also brings an updated version of LabComm which has seen some improvements in the way data is serialized. To accomplish this the LabComm and Firefly libraries were updated to work with VxWorks.

This enables low level control of a robot in ROS using the `ros2lc_bridge` package. Interest was then shown for a higher level interface against the robot, i.e., the possibility of using the Simulink part of ExtCtrl from ROS. This required a rewrite/update of the skele-

ton code, code generation, and Simulink Coder build process described in Section 2.4.2. The integration with external devices that used to be facilitated by an ORCA server was incompatible with the new version of LabComm used to connect to the IRC5. This interface was replaced with a Firefly server port that can be connected to a ROS system using a generated bridge, again by using the `ros2lc_bridge` generator.

5.2 Limitations

Due to yet unsolved problems with the computer platform to be used on the mobile robot the control code is currently written for standard Linux, with no Xenomai, RTnet or other real time patches. As shown in the histograms in figures 5.1, 5.2, and 5.3 this required the 1 ms upper limit of the round trip time of the IRC5 to be relaxed. This was done by inserting a one-slot buffer for responses from the external PC. This gives an additional period for the response to arrive. With a frequency of 250 Hz, this means an additional 4 ms for a total of 5 ms. This will likely decrease control performance for some applications using any of the rigid industrial robots available from ABB. In this case, however, we are using the Frida robot, which is not as rigid. More importantly, the end effector used in the current application has a large amount of compliance *by design*, as documented in [1, p. 37].

5.3 Benchmarks

Some comparisons of the time required for one round trip from the ABB controller to an external client is shown in figures 5.1, 5.2, and 5.3. The benchmarks consists of 2500 samples which corresponds to a 10 second window. The external computer is based on an Intel Core i7–4770 and is running a default installation of Ubuntu 12.04. No benchmarks are available for any sort of real-time client. The version of ROS used for the two latter benchmarks was Hydro.

5.4 Discussion

Figure 5.1 shows the round trip time using the modified system. Where the original system expected the round trip to be less than 1 ms, we achieve a mean of 1.03 ms. This is most likely the worst case scenario in the lab because the ABB Frida with 2×7 axes sends roughly twice the amount of data of any other robot with only 5–6 axes. The two distinct peaks and the large amount of jitter can be attributed to the fact that we were unable to run the experiment with real-time scheduling. The overall increase is likely because of this and possibly made worse by the use of Firefly and the higher level socket API used on VxWorks. More granular benchmarks will have to be implemented to determine this.

The difference between the C++ and Python implementations of ROS, shown in Figures 5.2 and 5.3 respectively, shows that ROSPY performs worse than ROSCPP, as expected, but in this case it has a lower variance.

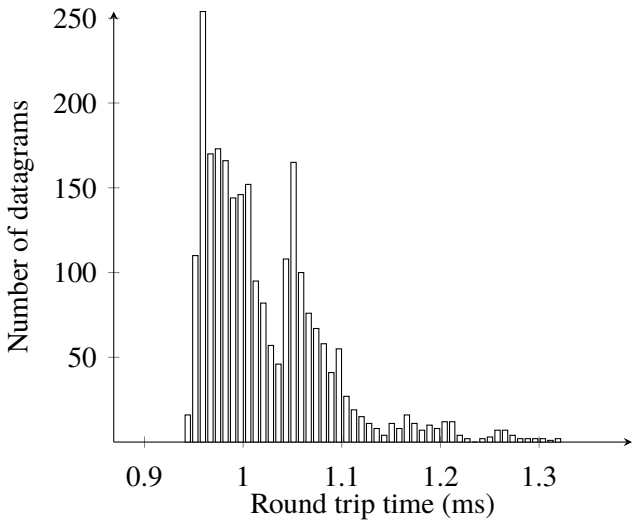

Figure 5.1: RTT between Frida and a non-ROS client. Mean: 1.03 ms, standard deviation: 0.06 ms.

These benchmarks are only supposed to illustrate the effect of using the bridge in conjunction with the two most common ROS client libraries. The most suitable setup for the future is shown in Figure 5.4.

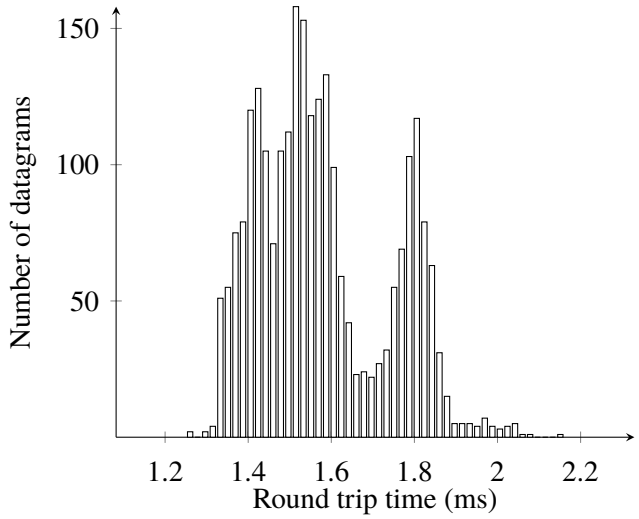

Figure 5.2: RTT between Frida and a ROSCPP client using a generated bridge. Mean: 1.59 ms, standard deviation: 0.16 ms.

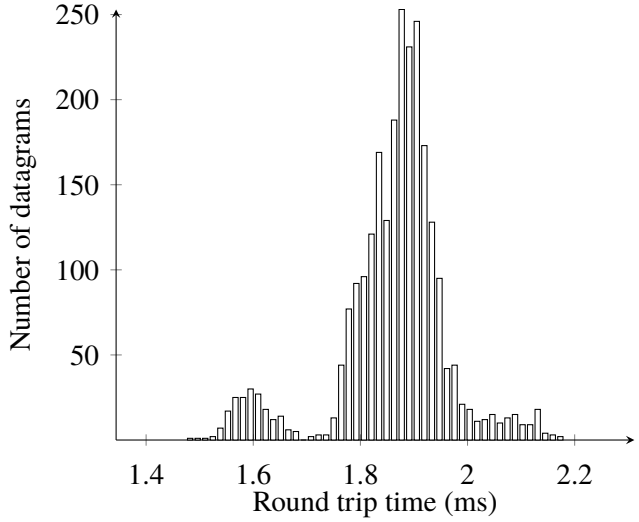

Figure 5.3: RTT between Frida and a ROSPY client using a generated bridge. Mean: 1.87 ms, standard deviation: 0.10 ms.

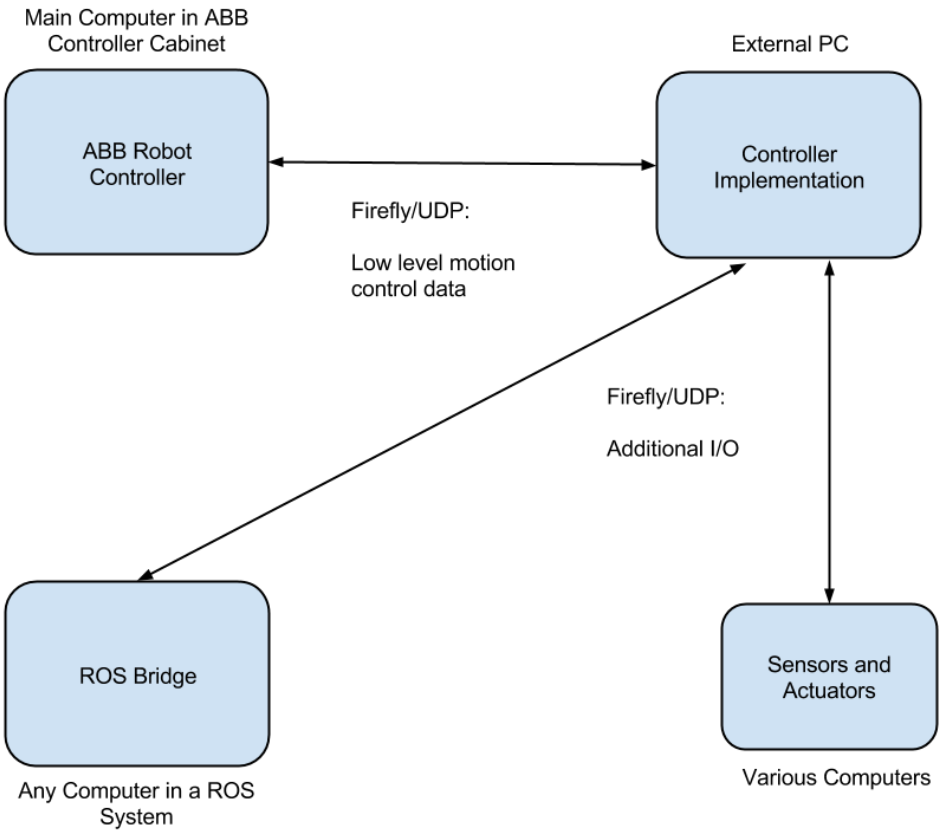

Figure 5.4: The proposed usage of the combined systems. The low level control is done independently of ROS which limits the control from ROS to whatever inputs are configured in Simulink.

Chapter 6
Future Work

To make full use of the work described in Chapter 5 in the larger context of the PRACE project some work is required. We describe, in deceasing order of importance, what we perceive to be the next tasks to focus on.

6.1 Controller Implementation

Some of the currently used controllers implement critical functionality in JGrafchart[1] instead of the Simulink model. That interface was replaced due to technical and architectural reasons. The coordination performed by JGrafchart will have to be moved, either to ROS or to the Simulink model itself. Alternatively, our work could be extended to accommodate a variety of interfaces.

6.2 Hand-Over Procedure

A big feature of the mobile robot system we have worked towards is the docking. Briefly put, the ABB robot is to be detachable from the mobile base in order for the two systems to work independently. This is the reason for some of the tangential content in Chapter 3. The hand-over procedure have to be researched once the integration has been successfully completed.

[1] A graphical state chart system mostly used for coordination.

6.3 Real-Time Controller on On-Board CompactPCI Computer

The real time support for the mobile platforms on-board computers is incomplete. Although a workaround was implemented and described in Chapter 5, improvements can be made. An effort was made to add additional benchmarks using the newly introduced `SCHED_DEADLINE` scheduler in Linux 3.14. This kernel built without issues but the C library had to be rebuilt in order to add the new system call `sched_setattr()` and this proved to be more of an issue. Running the controller with EDF scheduling should, in combination with some tweaks to the scheduling of Firefly's threads, improve performance.

Chapter 7
Conclusions

Two versions of the bridge generator was developed. The first generator was written in Python and generated a package using ROSPY, the Python implementation of the ROS client library. The amount of generated code was very small, thanks to the dynamic nature of Python. This bridge was not just an experimental stepping stone during development, but actually saw use by other people for successfully connecting existing software to ROS. Most importantly, these pieces of software could remain blissfully ignorant of the specifics of ROS communication primitives and concepts, a feature that previous attempts on bridging ROS with other systems does not provide.

Care was taken to make the bridge generator flexible and configurable so that incompatibilities and structural differences that might occur when adding, for example, additional sensors to a controller implementation, can be handled directly in the configuration instead of requiring "hacks" in the generated code. Such hacks would essentially defeat the point of being able to generate bridges quickly. This was also achieved; bridges can be customized at the time of generation.

As a next step we wanted to use ExtCtrl, firstly in conjunction with ROS and secondly on a PC that would not accommodate the existing implementation. For this task we chose to use the Firefly library, which is designed to make it easier to use LabComm in a more consistent way, especially over unreliable protocols. Some difficulties were encountered when porting Firefly to VxWorks due to subtle discrepancies in the POSIX-like API provided by VxWorks. The problems were solved after examining header files and discussing with people at LTH who were more familiar with VxWorks.

After some tweaks in its protocol, in addition to the porting work, Firefly was used successfully in generated bridges and both the VxWorks and the Linux components of a modified ExtCtrl. Currently, UDP was deemed a sufficient transport protocol but Firefly's modularity allows this to be switched if necessary in the future.

ExtCtrl was also modified for simpler start-up so that less coordination have to be performed by hand during, for example, docking and hand-over. For example, the modifications let the program install and uninstall itself into the control loop when clients connect

and disconnect. Previously, the user had to do this manually after connecting. By keeping such functionality that logically belongs to a program in that program, one can rely on fewer programs being started and controlled in the right order. This reduces complexity as well as the risk of failure during operation.

To use Firefly, currently completely implemented in C, and also to get the best possible performance from the bridges, the generator was extended to also generate C++. The generator itself is still implemented in Python because its flexibility and ease of use makes it a very good language for the kind of experimentation we have done. One can do a great many things in a concise and often clear manner. The one issue that made itself evident when the C++ backend came along was the lack of static typing. This has the potential of quickly making the code unclear and development error prone when the code size grows. After the completed implementation was refactored and cleaned up, development could continue at an acceptable pace.

The specific big picture goal of making it possible to connect the dual-arm manipulator and the mobile platform was reached. Due to time constraints however, the actual docking and handover scenario was not performed but the communication necessary for this was put in place and tested.

Appendices

Appendix A
Code

A large part of the code written or used during this thesis can be found in version control at LTH. In most cases the repositories contain examples of use and also documentation. Note that some of the documentation has lagged behind during the latter part of this thesis. This is planned to be rectified shortly, but in the case of any inconsistencies, the code naturally takes precedence.

What cannot be found in public repositories is the modified ExtCtrl implementation and some of the related bridge configuration files.

All software is described elsewhere in this thesis, so only our work is summarized below.

A.1 ros2lc_bridge

`http://git.cs.lth.se/robotlab/ros2lc_bridge`

The entirety of this project was implemented from scratch during this thesis.

A.2 Firefly

`http://git.cs.lth.se/robotlab/firefly`

This existing library was ported to VxWorks as part of the thesis. Some bugs in the protocol were found and corrected by introducing the concept of *auto-restricting* channels in the core library. The changes made during the thesis is currently only in the `vxworks` branch.

A.3 LabComm

`http://git.cs.lth.se/robotlab/labcomm-core`

The compiler was extended to emit C-functions for creating deep copies of samples. This was required in the C++ code generated by the `ros2lc_bridge` generator (it was already possible in python due to its garbage collection).

A.4 ExtCtrl

ExtCtrl was rewritten with Firefly and modified with relaxed timing requirements. This code is private to LTH due to heavy code reuse.

Bibliography

[1] Seventh Framework Programme. *PRACE, Annex 1 - "Description of Work"*, 5 2011.

[2] ROS about-page. `http://www.ros.org/about-ros/`. [Online; last accessed 2014-03-14].

[3] ROS 2.0 Messages + Serialization. `https://groups.google.com/forum/#!topic/ros-sig-ng-ros/c_obZW9Dr2A`. [Online; last accessed 2014-03-14].

[4] LabComm wiki. `http://wiki.cs.lth.se/moin/LabComm`. [Online; last accessed 2014-06-17].

[5] José Javier Colomer Vieitez. Managing real-time networking over switched ethernet. Master's thesis, Lund University, July 2011.

[6] Anders Blomdell, Isolde Dressler, Klas Nilsson, and Anders Robertsson. Flexible application development and high-performance motion control based on external sensing and reconfiguration of ABB industrial robot controllers. In *2010 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, May 2010.

[7] Anders Blomdell, Gunnar Bolmsjö, Torgny Brogårdh, Per Cederberg, Mats Isaksson, Rolf Johansson, Mathias Haage, Klas Nilsson, Magnus Olsson, Tomas Olsson, Anders Robertsson, and Jianjun Wang. Extending an industrial robot controller–Implementation and applications of a fast open sensor interface. *IEEE Robotics & Automation Magazine*, 12(3):85–94, September 2005.

[8] UPnP Forum. *UPnP Device Architecture 1.1*.

[9] Michael Jeronimo and Jack Weast. *UPnP Design by Example*. INTEL PRESS, 2003.

[10] Digital Living Network Alliance. *DLNA Overview and Vision*, June 2004.

[11] Jim Waldo. The jini architecture for network-centric computing. 42(7):76–82, July 1999.

[12] W. Keith Edwards, Mark W. Newman, Jana Sedivy, Trevor Smith, and Shahram Izadi. Challenge: Recombinant computing and the speakeasy approach. In *MobiCom '02 Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 279–286. ACM, 2002.

[13] David Svensson Fors, Boris Magnusson, Sven Gestegård Robertz, Görel Hedin, and Emma Nilsson-Nyman. Ad-hoc composition of pervasive services in the palcom architecture. In *International Conference on Pervasive Service*. ACM, July 2009.

[14] Google protocol buffers developer guide. `https://developers.google.com/protocol-buffers/docs/overview`. [Online; last accessed 2014-06-14].

[15] Cap'n proto. `http://kentonv.github.io/capnproto/`. [Online; last accessed 2014-06-14].

[16] ros_bridge. `http://rosbridge.org/doku.php`. [Online; last accessed 2014-03-10].

[17] rosbridge_suite. `http://wiki.ros.org/rosbridge_suite`. [Online; last accessed 2014-03-10].

[18] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011.

[19] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, RFC Editor, July 2006.

[20] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.