Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Stream Processing in the Robot Operating System framework

by

Anders Hongslo

LIU-IDA/LITH-EX-A--12/030--SE

2012-06-20

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Stream Processing in the Robot Operating System framework

by

Anders Hongslo

LIU-IDA/LITH-EX-A--12/030--SE

2012-06-20

Supervisor: Fredrik Heintz
Examiner: Fredrik Heintz

Abstract

Streams of information rather than static databases are becoming increasingly important with the rapid changes involved in a number of fields such as finance, social media and robotics. DyKnow is a stream-based knowledge processing middleware which has been used in autonomous Unmanned Aerial Vehicle (UAV) research. ROS (Robot Operating System) is an open-source robotics framework providing hardware abstraction, device drivers, communication infrastructure, tools, libraries as well as other functionalities.
This thesis describes a design and a realization of stream processing in ROS based on the stream-based knowledge processing middleware DyKnow. It describes how relevant information in ROS can be selected, labeled, merged and synchronized to provide streams of states. There are a lot of applications for such stream processing such as execution monitoring or evaluating metric temporal logic formulas through progression over state sequences containing the features of the formulas. Overviews are given of DyKnow and ROS before comparing the two and describing the design. The stream processing capabilities implemented in ROS are demonstrated through performance evaluations which show that such stream processing is fast and efficient. The resulting realization in ROS is also readily extensible to provide further stream processing functionality.

Acknowledgments

I would like to thank Fredrik Heintz for his insights and guidance throughout this entire process which has thought me a lot, Tommy Persson for our dialogues about coding in ROS and my father Terence for indulging my curiosity related to technical things and his unwavering support. I would also like to thank Daniel Lazarovski, Zlatan Dragisic as well as Patrick Doherty and the rest of AIICS and IDA.

Contents

Chapter 1

Introduction

The world changes all the time and the flow of information encompasses many areas of life and research. A continuous flow of information can referred to as a stream. These streams of information can describe many different things such as trending topics on the Internet, social media updates, financial instruments or data from sensors on a robot[7][10][14]. The focus of this thesis is stream processing within the specific domain of robotics although much of it is applicable in aforementioned domains too.

Figure 1.1: The most recent research platform from the AIICS group at Linköping University; the LinkQuad Autonomous Micro Aerial Vehicle from UAS Technologies.

DyKnow is a stream-based knowledge processing middleware used in autonomous Unmanned Aerial Vehicle (UAV) research[15]. One of the current research platforms involved is the LinkQuad quadrotor platform[14] shown in figure 1.1. UAVs have a lot of different applications spanning a number of different fields such as public service, military and aid during disasters. Among the scenarios explored by the AIICS research group at Linköping University are search and rescue missions where autonomous UAVs can be deployed to methodically search the nearby territory to locate humans in need of rescue during floods, fires, earthquakes or other dire scenarios. They are well suited for such assignments since they can operate in environments which might be hostile or unsuitable for humans, their senors enable them to perceive things humans might miss and their speed and mobility are also factors to consider. Locating humans affected by such disasters or even providing them physical objects such as aid packages or means of communication autonomously allows the humans involved in the rescue operation to spend their time more efficiently in a situation where each second counts.

Autonomous UAVs are not only fearless; they also lack the capacity to experience boredom which makes them suited for mundane assignments such as surveillance missions and traffic monitoring to detect traffic violations or accidents.

The goal is not to replace humans but to use autonomous UAVs as tools to help with certain tasks and thereby alleviate human workload. Just as with any tool it comes down to how you use it. Robotics can aid humans by doing tasks entirely autonomously but they can also help by notifying humans when their attention is needed such as when another human is in need of assistance.

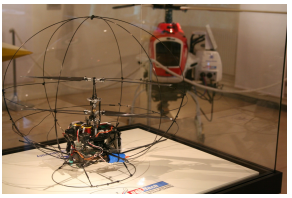
In such scenarios UAVs can be useful tools on their own and they can also sift through massive amounts of information and help to notify human operators where their attention is needed. This thesis is about processing the streams which contain such data to for instance use the resulting streams for execution monitoring or formula evaluation. This is done using methods from DyKnow in ROS (Robot Operating System).

DyKnow is a comprehensive framework which includes ways to deal with massive flows of information and derive structured knowledge from it which can span several abstraction layers, all the way from the sensory input to high level reasoning capabilities[15].

The ROS framework provides a large variety and quantity of practical tools for robotics research. The project is supported by many universities worldwide and there is ROS support for a number of robotics platforms[19].

(a) Yamaha RMAX

(b) PingWing



(c) LinkMAV

Figure 1.2: Other autonomous UAV platforms used by the AIICS group.

1.1 Motivation

Like ourselves robotics platforms have to handle massive amounts of streaming information. The challenge lies in sifting through these streams, isolating what is truly important and labeling the incoming data with familiar concepts to make sense of it as well as organizing it in a form that enables us to use the information to reason in a logical way about the world around us. The motivation behind this thesis is to provide useful DyKnow functionality in ROS to this end. DyKnow-concepts can aid with processing of data which can span several layers of abstraction, selecting relevant streams, merging streams and synchronizing streams. The last-mentioned is also essential for forming a state stream to model continuous time used to reason about the environment with metric temporal logic (MTL)[15] or qualitative spatio-temporal reasoning (QSTR)[18]; see sections 2.4.2 and 2.4.3 respectively. The stream processing also provides a foundation to further integrate functionality supported by DyKnow. The large amounts of information in ROS could benefit from a stream centric framework like DyKnow in a number of different ways such as making the framework more dynamic by providing operations on streams which can be configured and performed at runtime without changing the source code.

Stream processing is an integral part of DyKnow and the design and implementation of such is paramount to providing the functionality DyKnow has to offer in ROS. The stream processing functionality described in this thesis offers dynamic services at runtime to for example select relevant information in streams defined by constraints, merge streams containing contextually related data and synchronize streams into states. The selection and merge functions can be used for execution monitoring on robotic platforms and one of the motivations for synchronizing the streams is to provide state streams which can be used to evaluate MTL or perform QSTR.

1.2 Goal

The goal is to design and implement DyKnow-inspired stream processing in ROS. The solution should support runtime usage so that processes as well as users can specify and create streams they require to for instance perform execution monitoring or formula evaluation.

For service usage and certain autonomous operations a formal description of the streams content called a Stream Specification is required. The goal also includes a way to create these Stream Specifications using a language which has a syntax similar to common query languages.

The goal is to process data streams in ROS by performing operations such as select, merge and synchronize to form state streams. Select refers to being able to select specific data which is relevant based on a policy defining constraints which can be for instance temporal or based on matching a certain criteria. Merge refers to being able to unify data which relate to contextually similar concepts. The goal with regard to synchronizing is to swiftly form a steady stream of synchronized state samples based on the incoming data where each sample is synchronized around a certain point in time. The synchronization is to be done by matching the contents of time stamped data while taking into account if more data relevant to the current state can arrive in order to publish the states in a more expedient manner.

1.3 Outline

There are six chapters which compose this thesis; *Introduction*, *Background*, *Analysis and Design*, *Implementation*, *Empirical Evaluation* and lastly *Conclusions and Future Work*.

Background introduces the reader to ROS, DyKnow and related work.

Analysis and Design explores the differences and similarities of the two frameworks to lay the foundation for an architecture. A design for a DyKnow stream reasoning architecture and the stream processing module in aforementioned is presented. The stream processing operations are explained as well as the stream processing language SPL.

Implementation describes the realization in ROS based on the design. The messages and services used are explained in detail and examples of SPL are given.

Empirical Evaluation is the chapter where the implementation is tested and graphs are provided to explore the performance. The chapter mostly presents and dis-

cusses the overhead introduced by the stream processing operations.

The last chapter, *Conclusions and Future Work*, sums up the results and the process as well as deals with proposed research directions.

Chapter 2

Background

The *Robot Operating System* (ROS) is a modular open-source framework which is used and supported by a multitude of companies and universities worldwide. Architecturally ROS uses nodes which communicate using a publish/subscribe mechanism between them where nodes publishes messages of a specified type on corresponding topics which other nodes can subscribe to and thereby receiving the messages (2.1). A node in ROS can also provide services with predefined structures specifying the service's requests and responses.

The theoretical basis of the stream processing design in this thesis is based on *DyKnow* which is a comprehensive middleware framework based on streams and knowledge processing. It has mainly been used in the domain of robotics. Some of the concepts in DyKnow are streams which are continuous sequences of data elements, polices which specify the contents of such a stream, different kinds of knowledge processes which operate on streams as well as fluent streams which contain a sequence of discrete samples in order to approximate streams of data which might be continuous. A fluent stream policy is accordingly a set of constraints to specify the content of a fluent stream. DyKnow has been proven useful in several different contexts such as stream reasoning[12] and dynamic reconfiguration scenarios.

Stream processing has applications in robotics such as execution monitoring and providing state streams which can be used in stream reasoning to evaluate MTL or perform QSTR. Such stream reasoning is especially important since although *Data Stream Management Systems* (DSMS) can handle queries over streams, when it comes to complex reasoning they are somewhat lacking whereas most reasoners struggle with the rapidly changing data which is often involved in stream reasoning[10].

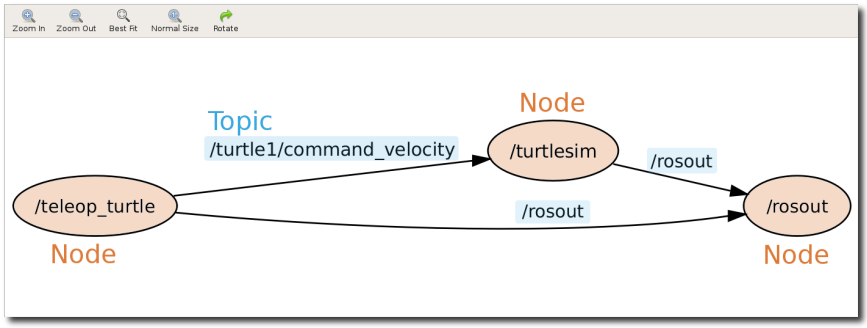


Figure 2.1: Tutorial example illustrating nodes and topics in ROS with the rxgraph tool

2.1 The Robot Operating System (ROS)

The Robot Operating System (ROS) is an open source framework for robotics software development with its roots at Willow Garage and Stanford University [19]. It consists of modular tools divided into libraries and supports different languages such as C++, Python and LISP. The ROS specification is at the messaging layer and consists of processes that can be on different hosts. Several universities from all over the world such as Stanford, MIT, Berkeley, TUM, Tokyo university and many others have put up ROS repositories contributing to this collaborative enterprise. Since it is a global collaborative open-source project in active development the code-base as a whole is in constant flux and lots of improvements are done for each new major release. Just during this project a couple of relevant changes have been made and in order to get the most up to date information about the things discussed here the reader is referred to the online documentation for ROS at www.ros.org where code, tutorials and examples can be found. The most recent version of ROS at the time of writing this thesis is Electric Emys.

2.1.1 Nodes

Computational processes in ROS are called *nodes*. A node can represent a wide variety of different concepts like for instance a sensor or an algorithm driven process.

2.1.2 Nodelets

Nodelets enables us to run multiple algorithms in a single process with zero copy transport between algorithms. There are however already some C++ optimizations in ROS which keep unneeded copy transport to a minimum within nodes.

2.1.3 Topics

Nodes communicate with a publish/subscribe pattern by passing messages to each other on *topics*. Each topic is referred to by a simple globally unique string such as for example uav1_altitude or uav1_front_laser_sensor. A topic can be considered to be a data stream and is strongly typed in the sense that you can only pass predefined message structures over a topic. For instance uav1_altitude might just pass a message containing a number whereas the uav1_front_laser_sensor topic might communicate a more complex message structure containing laser data. There may be several different publishers and subscribers to each topic.

2.1.4 Messages

The topics are strongly typed. An individual topic and its publishers and subscribers can only deal with a predefined message structure. These predefined messages are defined in .msg files which generate code for several programming languages upon compiling. There are a few primitive message types and ROS allows for the creation of more complex messages which can include several fields including arrays of messages. There is no explicit support for recursive messages (although workarounds such as publishing addresses might be possible for embedded systems).

Here is a simple example to illustrate what a message can look like:

contents of *UAV*.msg:

```
Header header
      uint32 seq
      time stamp
      string frame_id
uint32 id
string uav_type
uint32 alt
uint32 spd
```

In the example above there are 5 fields called header, id, uav_type, alt and spd. Their respective type is listed to the left of the field name. The type Header is not a primitive field: it is a message as indicated by the capital H and illustrating the fact that it is possible to have messages composed of other messages and so on with even deeper nesting.

Another feature is that a field can be an array of messages such as:
UAV[] uavs
where uavs would be an array of the message type UAV.

2.1.5 Services

Services in ROS work according to the familiar request/response pattern common in computer science and are defined by .srv files. Each .srv file has the following format where request and response can be zero or more message classes:

Example.srv:
request
—
response

2.1.6 Synchronization in ROS

There are several different ways to go about synchronization and some methods are already available in ROS. The package message_filters has a time synchronizer based on templates which is available for C++ and Python. A motivating example for the synchronization done by this package is to synchronize messages from two different cameras to provide stereo vision so the robot can get depth in its vision like humans can see in 3D. The existing synchronization functionality and how it fits the needs of stream processing is discussed further in section 3.5.3.

2.1.7 Development, Tools and Extensions

There are quite a few tools included in the ROS platform so only a few will be mentioned here. One of these is rxgraph which visualizes the currently running collection of nodes and topics. Yet another one is roslaunch which makes it possible to first specify complex setups of nodes and topics along with parameters and attributes in XML and then run them. Due to the efforts from the team at

Willow Garage and because of how active the ROS community is the scope and functionality of ROS keeps increasing. Functionality such as message serialization was not part of ROS in the early stages of this project. Some relevant packages are currently only at the experimental stage such as rosrt (ROS real-time) which has publishers and subscribers which are better suited for real-time processing, yet another package very much still in development worth mentioning is rostopic which allows for some dynamic behaviors, filtering of messages at real-time and easy accessible information about topics and their messages using Python. So as the work on the ROS framework progresses it will probably gain further functionalities which make it an even better platform for the kind of dynamic stream processing discussed here. Furthermore it is important to note that since ROS is an open-source project there are also lots of additional stacks created by robotics companies, passionate individuals and prestigious universities all over the world.

2.2 Stream-Based Knowledge Processing

Heintz et. al. describes both a general ***stream-based knowledge processing middleware framework*** and a concrete instantiation of such a framework called ***DyKnow***[15][13][14].

The general stream-based knowledge processing middleware framework contains the concepts *streams*, *policies*, *knowledge processes* and *stream generators*. These concepts are mirrored in a concrete instantiation, DyKnow, wherein streams are specialized as *fluent streams* and the knowledge processes as *sources* and *computational units*.

2.3 DyKnow

DyKnow is a stream-based knowledge processing middleware framework and a concrete instantiation[15] of a generic stream-based middleware framework discussed above in 2.2. Knowledge processing middleware is defined by Heintz as a systematic and principled software framework for bridging the gap between the information about the world available through sensing and the knowledge needed to reason about the world[12]. Much as our own eyes can deceive us, a robot's sensors does not entail everything about the surrounding environment. This gap between the real world and the sensory-based internal model used to make decisions will be referred to as the sense-reasoning gap.

It is not only the limited and noisy data obtained from sensors which has to be abstracted into knowledge. A stream in DyKnow can be formed from a multitude of

inputs spanning different layers of abstraction. For instance in robotics we might have dozens of sensors with somewhat fallible information yet also be connected to the Internet or databases with more specific knowledge. DyKnow supports integration and processing of sources with varying degrees of abstraction and bottom-up as well as top-down model-based processing of them. In a system where sensor data is abstracted into knowledge there is of course a large degree of uncertainty since previous hypotheses might get disproved by new data. For instance if an autonomous robot gets very limited sensory data about an object it might at first label it as a as a small building whereas this hypothesis needs revising when new sensory input associates the object with features disproving the old model, such as speed or altitude, then the autonomous robot should use a different abstraction to reason about the object since it isn't a building. The uncertainty which is inherent in dynamic environments such as the real world and the sensory-based models used to reason about the world makes it important to support flexible configuration as well as reconfiguration to accommodate for changes. These changes can derive from many different sources, not only new sensory data from the robot itself as previously mentioned. The robotics platform could have remote links to other robots, off-site information on a server or in the cloud. Flexibility and the ability to reconfigure can be used to reduce the computational burden or to make the system as a whole more robust since during search and rescue missions robotics platforms could be exposed to harsh conditions where such reconfiguration could be used to maintain operational status.

DyKnow represents the state of the system over time and its environment with streams and therefore they are an integral part of the framework. Each stream's properties are specified by a declarative policy.

The current implementation of DyKnow is built upon the Common Object Request Broker Architecture (CORBA) standard and has support for chronicle recognition and MTL with formal semantics. As an example a UAV in a traffic monitoring scenario is then able to recognize events such as whether a car halts at a stop-sign or overtakes another car.

2.3.1 Streams

A stream consists of continuous data from one or more sources. Such a data source for a stream can be a number of different things, for instance a hardware sensor, information retrieved from a database or generated by a computer program[1].

Stream: A stream consists of a continuous sequence of elements and the formal definition also includes that each element contains information about its relevant time.

2.3.2 Policies

A policy specifies the requirements on the contents of a stream. For instance we may want to make sure that the messages have a certain frequency, maximum delay or relative change compared to the prior value.

2.3.3 Knowledge Process

A knowledge process is a quite broad concept and refers to basically any process that operates on streams. There are different process types: *primitive processes* such as sensors and databases, *refinement processes* such as filters, *configuration processes* that can start and end other processes or finally *mediation processes* that can aggregate or select information from streams.

2.3.4 Stream Generators

A stream generator is able to provide an arbitrary number of output streams that adheres to given policies. For instance we might have a knowledge process in the form of a sensor (a primitive process) and we have other refinement processes that need the sensor data. But one refinement process might need data every 10 ms while another one only needs it when the value has changed by a certain amount.

There are two classes of knowledge processes in DyKnow: sources and computational units. Sources correspond to primitive processes (i.e. sensors and accessible databases) and computational units correspond to refinement processes (i.e. processing fluent streams).

The current implementation of DyKnow is as a service on top of CORBA which is an object-oriented middleware framework often used in robotics. The DyKnow service in CORBA uses the notification service to provide publish/subscribe communication.

2.3.5 Features, Objects and Fluents

DyKnow uses the concepts of Objects and Features to model the world. Objects describe a flexible type of entity can be abstract or concrete and their existence does not have to be confirmed; hypothetical objects are also important when dealing with the uncertainty of the UAV domain. Features describe properties in the domain with well defined values for each point in time. An example in the UAV

domain would be objects such as UAVs and cars which all have features like speed, location relative position.

2.3.6 Value

Features have values for each point in time which describe their current state. A value can be one of three things; 1) a simple value in the form of a object constant, time point or primitive value 2) the constant no_value, 3) a tuple of values. Some examples of such values are the color of the UAV, the missions start time, the current altitude, observed dronts (no_value) and observed cars (which can be a tuple of values).

2.3.7 Fluent Stream

Both sources and computational units can be asked to provide us with a *fluent stream* which is an approximation of the previously mentioned streams (2.3.1). In these fluent streams the elements are called samples since that's what they are: samples to provide us with an approximation of the more abstract concept of a stream. There are two different time concepts that are central in DyKnow: available time and valid time. Available time is simply when the sample/element is available as elucidated in 2.3.1. Valid time on the other hand is when it was valid, for instance if we have a sensor the valid time of this sensor is when the sample was taken and the available time is when it arrives to a computational unit to calculate some formula which is dependent upon it.

2.3.8 Sources and Computational Units

Sources tell us the output of a primitive process at any time point; we might for instance want to know the output from a laser sensor or a GPS.

Computational units on the other hand refer to processes that compute an output given one or more fluent streams as input. Obviously this covers a broad range of functions; anything from simple arithmetic to calculating complex algorithms. Examples include filters, transformations and more general mathematical operations.

2.3.9 Fluent Stream Policies

A policy is a set of constraints on a fluent stream. For instance we might have a maximum delay on the samples we want to use or perhaps we want to make sure that the samples arrive in the correct temporal order.

2.4 Related work

Stream processing is useful in the area of stream reasoning and there is some overlap there when it comes to related areas. Stream reasoning query languages can express formulas which are to be evaluated over streams. There are several different sorts of stream reasoning which can be useful within the domain of robotics such as Metric Temporal Logic progression and Qualitative Spatio-Temporal Reasoning.

2.4.1 Stream Reasoning

A stream can be defined as a collection of data values presented by a source that generates values continuously such as either a computer program or a hardware sensor[1].

Continuous and time-varying data streams can be rapid, transient and unpredictable. These characteristics make them unfeasible for use with a traditional database management system (DBMS). A traditional DBMS is unable to handle continuous queries over the streams and lacks the adaptivity and handling of approximations[3]. One could take a snapshot of the state of the data and use a reasoner on that static model but without further constraints it is uncertain how useful the result will be since we might have gotten new data right after the snapshot was taken. Moreover that snapshot would not tell us anything about the system's history over time.

Data stream management systems (DSMS) are able to evaluate queries over streams of data but they are not able to handle complex reasoning tasks. Reasoners however are capable of complex reasoning but do not support the rapid changes that occur with vast amounts of streaming data[9]. The integration of data streams and reasoners in the form of stream reasoning provides the capabilities to answer questions about changing data.

The applications for stream reasoning are quite wide in scope. These include monitoring and reasoning about epidemics, stock markets, power plants, patients,

current events or robotic systems.

Querying Streams

There are several query languages which deal with performing queries over streams which can involve very rapidly changing data. There are several languages related to this such as StreamSQL, CQL, CSQL and C-SPARQL. Continuous Query Language [2] (CQL) is a language used for queries over streams in for instance DSMS applications and it has been backed by Oracle. StreamBase StreamSQL is a competing standard in that domain[16]. There is also Continuous SPARQL (C-SPARQL) which is a language for continuous queries over streams of Resource Description Framework (RDF) triples and it has been used in stream reasoning research [4][5].

2.4.2 Metric Temporal Logic Progression (MTL)

As an extension of propositional linear temporal logic with discrete time-bound temporal operators MTL allows us to set specific temporal boundaries in which formulas must hold[20]. Since execution logs can become very large if we have large amounts of streaming data these temporal constraints make MTL a good fit for real-time monitoring. One real world example could be that the reactor temperature may not exceed a certain number of degrees for more than five seconds. An example more relevant to the domain of robotics is when we are executing a task plan on a UAV. Since it is unfeasible to simply assume that no failures can occur during the execution of this task it is prudent to monitor it with conditions and notify when something doesn't go according to the task plan. Execution monitoring with MTL informs us when the UAV runs into issues thereby giving us more time to adjust the task plan accordingly.

2.4.3 Qualitative Spatio-Temporal Reasoning (QSTR)

Extensions to the reasoning done in DyKnow to include spatial reasoning has been done by Lazarovski[18]. It uses Region Connection Calculus 8 (RCC8) to provide qualitative spatio-temporal reasoning by progressing over synchronized state streams such as the streams created by the processing functionality outlined in this thesis. RCC8 describes 8 basic relations between areas such as for instance disconnected, equal or partially overlapping. More about how this kind of reasoning works in ROS in the next chapter.

2.4.4 Cognitive Robot Abstract Machine (CRAM)

Beetz et. al. at the Technische Universität München (TUM) has developed a software toolbox called CRAM (Cognitive Robot Abstract Machine) which enables robots with flexibile lightweight reasoning[6]. CRAM is implemented as a stack with a ROS interface. Two essential components of CRAM are the CRAM Plan Language (CPL) and the knowledge processing system KnowRob. The expressive behavior specification language CPL makes it possible for autonomous robots to execute and also manipulate and reason about its control programs.

A difference between the work done at TUM and what is discussed in this thesis is that the reasoning in CRAM is done in batch-mode and not continuously which is basically the difference between reasoning and stream reasoning.

Chapter 3

Analysis and Design

The common ground between DyKnow and ROS outlined in the previous chapter is part of the reason behind a few of the design decisions in this chapter. For instance the similarities between nodes in ROS and knowledge processes in DyKnow indicates that efforts might better be spent elsewhere than to contrive things by including a similar design concept in ROS. Instead the design philosophy in this thesis has been to focus on the capabilities of DyKnow which could add value to ROS and then investigate how to design it in practice. From this point of view the differences between them are very important since the differences describe not only the potential value of DyKnow but also differences which have to be reconciled to run it in practice. Due the scope of DyKnow and its general nature the parts which should provide valuable additions to ROS supersedes the scope of this thesis although the design and implementation of a few will of course be described. This chapter describes some of the design considerations taken when integrating the two. An overview of a system for evaluating metric temporal logic is also provided in the chapter as an example of stream reasoning in ROS.

There are similarities between them; most notably between streams and topics as well as between knowledge processes and nodes. Of course there are also differences such as the central role of policies in DyKnow and the strongly typed topics in ROS.

Adhering to the design paradigms in ROS means a modular design divided into nodes with the code separated into packages. A stream reasoning architecture that contains several modules has been designed with this in mind. The stream processing is contained in one of these modules and has its own package. The stream processor can be ordered to create streams which are described by stream specifications. The stream specifications contain the policies, constraints and operations which are to be performed. The main stream processing operations are

select, merge and sync.

Since stream specifications have a programmatic syntax with a structure designed for machines rather than humans there is a need for something more user friendly which can make the stream descriptions more legible. The Stream Processing Language (SPL) is designed with this in mind and bears some resemblance to languages such as SQL whose operators users are more likely to be familiar with. SPL allows for a concise description of an entire stream and its policies and the SPL expression can be translated into a stream specification the stream processor can use directly.

3.1 DyKnow and ROS

In this section we will take a look at how some concepts in DyKnow correspond to the framework provided by ROS in order to assess similarities as well as differences. Many of the factors mentioned here have been taken into consideration with regard to the overall high-level design of the stream reasoning architecture outlined later in the chapter.

3.1.1 Streams (DyKnow) and Topics (ROS)

The streams in DyKnow bears many similarities with the topics in ROS. Both concepts refer to continuous data values from a source as discussed in the previous chapter so topics are also streams of data. Fluents however are more strictly defined by policies and consequently a topic can not automatically be seen as a fluent without the messages published upon that topic conforming to some constraints.

3.1.2 Knowledge Processes (DyKnow) and Nodes (ROS)

There are similarities between the two concepts since Knowledge Processes (KP) in DyKnow and nodes/nodelets in ROS are very versatile and can span many different layers of abstraction. A KP/node can represent for instance a sensor, a process acting upon sensor data or a process acting upon inputs from several different other KPs/nodes.

3.1.3 Setting up processes

Both DyKnow and ROS have tools to set up networks of processes; KPL and roslaunch. KPL focuses on the declarative specification of the network created by using formal semantics with explicit constraints while roslaunch creates the network by using information about nodes, parameters and attributes as specified in an XML file.

3.1.4 Modularity

The division into nodes done in ROS and the focus on flexibility in DyKnow are a couple of the reasons why a modular design would make sense when combining the two. Having separate ROS nodes for the DyKnow capabilities keeps concepts separate and makes it possible for the parts to provide functionalities to the ROS community on their own. The polar opposite of this would be to have a closed system for all the DyKnow capabilities and only providing one interface to ROS. Such a closed system might offer better performance if everything runs on the same embedded platform due to lesser use of bandwidth in ROS. The lack of flexibility of such a black box is its downfall though. Having a more modular design could also make it easier to run it on a platform with access to distributed computing power which is one of the strengths in ROS.

3.1.5 Services/Parameters

There are several ways to provide functionality in ROS during runtime. Here are a couple of the ways to do it: passing parameters to a node when you start it is one way and having a service node running which uses a request/response mechanism is another. Using one of these doesn't necessarily exclude the other yet comparing the two to figure out the primary way to communicating between the modules seems prudent.

Key components related to stream reasoning in DyKnow are designed as services; both in order to maintain the modularity of the DyKnow design and also since it goes well with the design philosophy in ROS.

An elaborate design to set up parameters to start nodes doesn't seem to provide enough advantages compared to having nodes which provide services. Services seem to be the favored design for such an endeavor in ROS and the separate specifications of services provided in .srv files makes the services readable, formal and usable whereas a design with parameters would require separate documentation for parameter-code which would be less integrated.

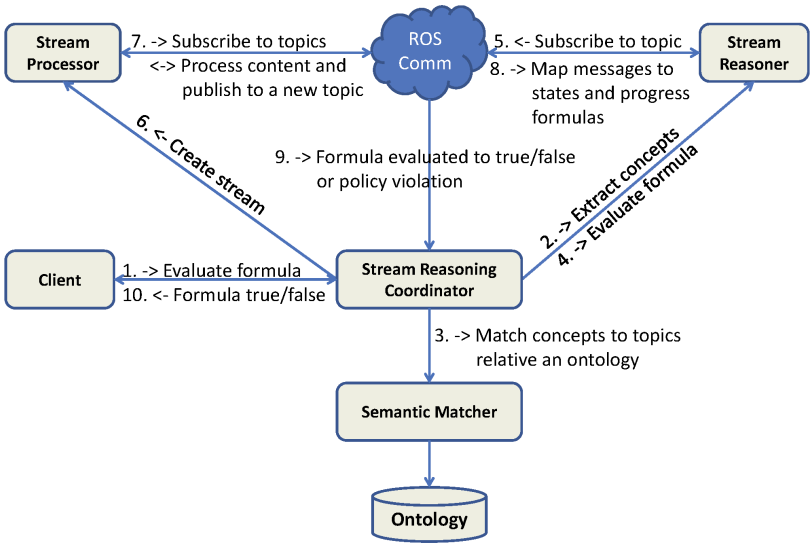


Figure 3.1: Overview of the current DyKnow stream reasoning architecture implementation in ROS

3.2 A Stream Reasoning Architecture for ROS

Stream processing is a necessity for the formula progression to work yet it can also be useful on its own where for instance a stream must adhere to specific constraints or synchronization of sensor data is needed. One scenario would be when data from different sensors on an autonomous robot has to be synchronized to form a stream of states used to reason about it on a higher abstraction level. While ROS already supports some constraints and synchronization in their message_filters it is by design more focused on discrete updates rather than the creation of streams. For instance the current ApproximateTime synchronization policy in ROS uses each message only once whereas in stream based reasoning it could very well be more appropriate to use a message again as a component of a state rather than wait too long for the next message to arrive.

As can be seen in Figure 3.1 the current architecture consists of several parts; the stream reasoning coordinator with semantic matcher and ontology, the stream reasoner and the stream processor. Together they can be used to for instance evaluate spatio-temporal formulas with semantic matching. The focus of this thesis is on the stream processing. The Stream Processor is an integral part

for the reasoner to evaluate formulas dealing with temporal logic since it needs continuous streams providing information about the relevant features to progress over.

As much of research today the Stream Reasoning Architecture is a part of a collaborative effort. The overall design of this architecture and the stream processing are parts of the work done for this thesis and it has been in close collaboration with Fredrik Heintz whose expertise and vision for DyKnow have been essential. The Semantic Matcher and ontology parts have been researched in detail by Dragisic[11] and the Stream Reasoner is based on work by Heintz and Lazarovski[12][18]. Their contributions are essential for the Stream Reasoning Architecture as a whole.

In Figure 3.1 the client can order the stream reasoning coordinator to evaluate a spatio-temporal formula. Concepts related to the terms in the formula are subsequently extracted. Semantic matching using the ontology is then used to find out on which topics information relevant to these concepts can be found in our system. The stream reasoner is then told about the formula it is about to evaluate and sets up a subscription on the topic where it wants the state stream with the necessary data. The stream processor is then given the task of creating policy-based subscriptions of the incoming topics, selecting relevant data and as needed properly label, merge and synchronize the data to finally publish it on the state stream topic. The stream reasoner can then progress over the state stream it has subscribed to in order to evaluate the formula. Finally the answer is given to the client. A stream processor node can also create streams on demand by being called directly by the client since it provides a service as defined by ROS.

An example where we evaluate a metric temporal logic formula can be to ask whether or not all of our unmanned aerial vehicles always have an altitude of over 2 meters. In order to answer this we first have to find out where the required information about the altitudes can be found and isolate these features. Then each altitude gets a corresponding fluent and these are then synchronized into a state stream. The grounding context contains necessary information such as the fluent policies and synchronization method. The state stream is used by the reasoner to keep the symbol table updated and evaluate the formula.

3.2.1 Stream Reasoning Coordinator

As the name suggests it coordinates the modules to perform the given tasks through communication between them and also between it and the client. It relays information mostly through services defined in ROS as described in section 2.1.5.

3.2.2 Semantic Matcher and Ontology

The semantic matcher service uses one or more ontologies to perform semantic matching on symbols in formulas to the content of streams. By doing this the system can be aware where relevant information is to be found when a query is posed in the form of a formula to evaluate[11]. This information can then be relayed to the stream processor which sets up the relevant streams and synchronizes the data so the reasoner can evaluate it.

3.2.3 Stream Processor

The stream processor's purpose is to select, properly name, merge and synchronize relevant data to create state streams. A modern robotics platform is a complex system with lots of streams on different layers of abstraction and the stream processor is able to select relevant data from these layers and process it accordingly by merging streams while also offering high-performance synchronization of multiple streams.

Being able to process streams in this manner has several different applications such as MTL evaluation. When evaluating formulas the reasoner is fed the relevant data on a dedicated topic with a uniform type so it does not have to handle irrelevant information nor subscribing to multitudes of topics with different types since all of this is handled by the stream processor. Furthermore because of the very high data rates on some streams it is very important to select only what is relevant since the reasoner would in many cases be overwhelmed by data otherwise.

The stream processor will be dealt with in more detail since it is one of the focal points of this thesis.

3.2.4 Stream Reasoner

The DyKnow stream reasoner does the final evaluation of the formula by progressing over the provided state stream and checking whether or not the formula is true. The reasoner is capable of both temporal reasoning [15][12] as well as spatial reasoning and Lazarovski[18] has implemented it so the stream reasoner can be accessed as a service in ROS.

State synchronization is important for several reasons for instance when the stream reasoner evaluates formulas the continuous temporal dimension is modeled as a periodic stream of discrete synchronized states where each sample in the stream is considered valid until the next sample and therefore it wants a steady stream of

these.

A fitting sample period is important to decrease the likelihood of missing vital information between the samples. For instance if one where to evaluate if the altitude of an autonomous UAV was below 5 m/s under a time period we might very well miss a slight dip below five if the sample rate was very low whereas the likelihood of that happening with a stream with a high sample rate is very low.

This thesis will deal with a few different ways of synchronizing streams below and in the next chapter.

Valid Time, Available Time and Sync Time

In larger and more distributed systems the latency between when the information is produced and when it arrives to the stream processor can be significant. Unless otherwise stated a value will be considered valid until a new one is produced.

Valid time is when the value in a message is considered valid, for instance the time point when a sensor measures something. Available time refers to when the value is available to the stream processor. In other words when it arrives to be processed. Sync time or synchronization time refers to a point in time the stream processor will create a synchronized state from two or more values. The next sync time will be the previous sync time plus the sample period.

A simple example is when the intent is to evaluate if UAV 1 is faster than UAV 2 under the upcoming ten seconds, in that case each sample in the resulting synchronized state stream will contain both the speed value for UAV 1 and UAV 2 for a specified sync time and the stream will consist of such samples with incrementally higher sync times for the entire ten second period.

Expected Arrivals

In order to publish a state as soon as possible without doing so prematurely it is very important to have a sense for if any more relevant information is due for this state. Delays should be kept minimal yet a slight wait to get the latest values is often good, the tricky part is to know when to wait and when to publish. Publication should be done when no more information regarding that sync time is due to arrive in time. We can be sure this is the case when sync time plus the duration of the maximum delay, as defined by the policy, has passed. There are situations where it can be deduced sooner though.

For instance if the current sync time is 2230 ms and we just got messages with

valid times of 2190 from both of the topics which affect the states at about 2000 ms. If no more messages are due to arrive in time there is no use waiting. If the incoming topics only publish every 100 ms we can be sure no more information relevant to 2230 will arrive, hence we can publish the final state for 2230 ms sooner at about 2000 ms, since the next incoming messages are not going to be relevant for that sync time.

When using the merge operation in stream processing several topics can affect the same field in a state and the topics can have different rates.

Sample Period Deviation

Allowing a sample period deviation can be beneficial in some cases since it allows taking when messages arrive into account when deciding sync times. This would mean that the period between each message would not have to be fixed but instead be in a certain interval. Since the states in the stream can be affected by several topics with different rates it could sometimes be beneficial to choose each specific sync time from an allowed interval to better match the incoming messages relevant to the interval. One suggested way to do this would be by choosing a sync time which minimizes the sum of temporal differences between it and the valid times of the samples which are expected to be available. Essentially this would mean calculating the geometric median (Fermat-Weber point) which minimizes the delays since this problem can be viewed geometrically as lines modeling the temporal domain and the distance between the points in time are the delays. A way to solve this is with for instance Weiszfeld's algorithm[8]. A downside to allowing a sample period deviation is that this approach would introduce significant computational overhead. This might be adequate if the incoming topics all have a fairly low rate although for very rapid streams of data it could be problematic. The method is however worthy of consideration as an optimization if the computational resources are available.

3.2.5 Notes About Programming Languages

Since ROS supports several programming languages such as C++, Python, LISP and Java the choice of language warrants some discussion beforehand. Another thing to note is that communication between nodes is the same no matter which language they are implemented in since they use topics and messages. The choice of language is therefore something best done for each node independently based on the requirements and possible dependencies. The semantic matcher for instance uses Java extensively due to external libraries. At the point of writing C++ and Python are arguably the languages ROS has the most support for and both warranted some consideration. A few of the reasons to go with Python are: a dynamic

language would allow for more dynamic type handling at runtime, although somewhat subjective some might argue it offers superior readability and allows for fast prototyping. The most apparent downside to using Python is that C++ is very likely to result in faster running code although this obviously depends somewhat on how the code is written and choice of compiler. The previous CORBA implementation was done in C++ and while code from it has been reused in the reasoner module, the stream processing code has been rewritten from the ground up to make it closer to the ROS framework. Some prototyping has been done in Python yet most of the final code base is C++ centric due to performance priorities.

3.3 Stream Processing

There are of course a multitude of different ways one could implement anything in ROS due to the modularity and flexibility of it as well as how it supports several programming languages. This section will discuss a couple approaches and their advantages and disadvantages in comparison with the current implementation explained in the previous chapter.

The ease of doing stream processing at runtime instead of having to change the source code to perform the related is also something which has been taken into consideration when designing the stream reasoning architecture and the stream processor component.

3.3.1 Type Handling

ROS is strongly typed in the sense that a topic can only publish messages of one type. The previous implementation of DyKnow in CORBA made frequent use of the 'any type'. In order to reconcile this difference the messages subscribed to in the stream processor are converted into a universal type during processing.

The states in the resulting streams can be composed of fields from several different message structures which has vast implications on the possible combinations of the type on the state stream topic. The steams are created at runtime yet messages in ROS have to be defined in advance. Compiling all the possible combinations is not an option nor is splitting the state into separate messages on separate topics since it not only introduces a lot of overhead; it also negates the purpose of forming the states in the first place.

Serialization

ROS 1.1 introduced ros::serialization for roscpp which uses C++ templates to serialize/deserialize roscpp messages as well as other C++ types. Once serialized selecting relevant fields with message introspection is not as easy as in Python though. One could however select fields based on the original message, then serialize the selected parts and keep them in a container in the buffer. The container in this approach could then contain related information such as timings relevant for synchronization. Publishing a state composed of selected fields from messages poses somewhat more of a challenge however since the serialized message is not an ordinary ROS message type. Receiving the address to the serialized messages in the subscriber would work if one assumes that the publishers and subscribers share memory however if one wants a more distributed system spanning several platforms with separate memory it poses more of a problem.

The current design converts incoming messages into organized structures where the data is represented as string fields in a class based hierarchy. The solution is similar to ros::serialization and created before it was released out of necessity. It is possible that ros::serialization could be a viable alternative with some modifications although this design is very good when introspecting converted messages to select relevant content. The current design handles nested structures and composed structures by naming the path to each subfield and the final converted class is composed out of them.

3.3.2 Nodes and Nodelets

When it comes to the usage of nodes and nodelets there are a few factors to consider such as performance, how clear the design will be and issues such as thread-safety. Because of the C++ optimizations which already keep costs related to copy transport low within the stream processor and because of the thread safety aspects it was decided to have the stream processor as a node.

3.4 Stream Specifications

What a stream generator and resulting stream will consist of has to be decided somewhere along the way and the message structure in which most of this is done is aptly named the stream specification. A stream specification contains all the information needed to set up a stream generator such as applicable constraints, the sample period of the resulting stream, information about the topics which are relevant for the stream, which fields are relevant, optional renaming data, which

fields to merge and which to synchronize and more. How all of this is implemented is discussed further in the next chapter.

3.4.1 Stream Constraints

The stream constraints specify the relevant times for the stream, the desired sample period, the maximum delay between valid time and arrival time, if the stream is to be ordered on valid time as well as if the valid times for the fields can have the same valid time.

Constraints such as the maximum delay are imposed by imposing them on the incoming messages to ensure that they are enforced. If messages are not ordered on valid time it can pose a problem since it would then be impossible in some cases to know that we have received all the relevant information for a point in time.

There is also an optional sample period deviation. The purpose behind this is that it could sometimes be beneficial to allow the stream generator itself to have some leeway when it comes to deciding the actual sync times in the stream rather than just always having the same static periodicity.

3.5 Stream Processing Operational Overview

Figure 3.2 shows the design of a stream generator. Incoming messages on one or more topics are put through a select process outlined in 3.3, then merged and synchronized. How an individual stream generator is set up does however depend on how the stream is specified; some might contain just one select process where others can consist of several select processes merged or several synchronized features from several merges all of in turn consist of a multitude of select processes.

3.5.1 The Select Process

Selecting messages on a topic which fulfills certain criteria or passes given constraints is useful in number of different applications such as runtime monitoring.

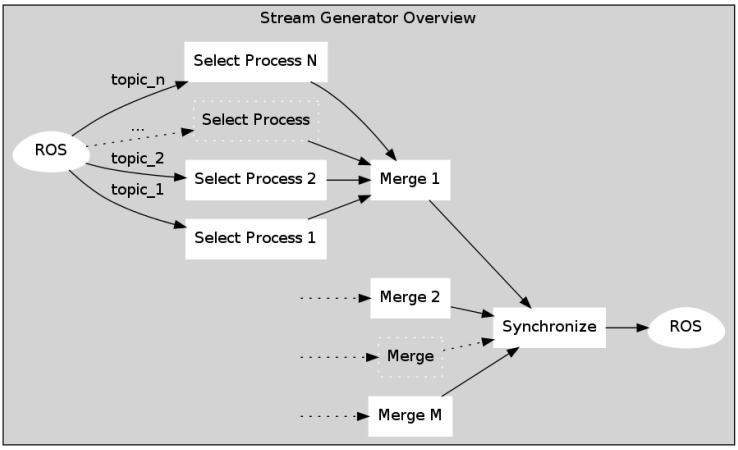


Figure 3.2: An overview of a stream generator performing its operations.

Select Example

For instance we might want to select the altitude of the UAV with *id* 1 from the field named *alt* in certain messages on a topic named *info_about_uavs* to get a resulting stream with a sample period of 200 ms. The topic *info_about_uavs* could publish messages of a type which contains a lot of fields irrelevant to our purposes, it could have a very high sample period and furthermore it could contain entire messages which are irrelevant since they are about other UAVs. Therefore it is useful to be able to select what is important in this stream of information.

Programming Languages

The ease of implementation as well as performance when it comes to this feature is heavily dependent on which language is chosen. Python offers great flexibility and ease of use since getting representations of messages in the stream and subsequently manipulating them is easily achieved. The command prompt tool rostopic demonstrates this functionality when called with the filter parameter and a Python expression explaining which messages to select. As the documentation says performance poses an issue under more stressful conditions though which is inherent in the use of a dynamic scripted language compared to C++ which is compiled and much closer to the hardware than Python.

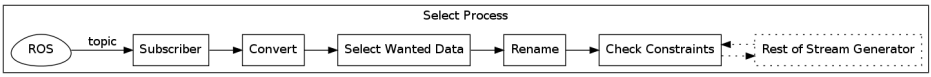


Figure 3.3: The select process describes what happens to an incoming message on a relevant topic. First the subscriber receives the message. Since the topics are strongly typed in ROS the message is of a certain message type and to solve the issues previously discussed in section 3.3 the messages are converted into a universal type before further processing. Then the relevant fields are selected and optionally renamed to something more suitable. If the incoming data passes the relevant constraints it is then sent further in the stream generator. One single incoming message from a topic can of course contain several fields which are relevant to the states in the resulting stream, one way of dealing with this is to send the same original message to several select processes each in which a different field is selected and another way is to select several fields in the same select process.

3.5.2 Merging Streams

Merging streams is useful when the streams contain information which in this specific situation describes similar features. Fields found in different messages on different topics might hold the same relevance in a situation and in such scenarios it makes sense to be able to merge these fields into one in the resulting stream.

Merge Example

For instance if an autonomous UAV has two non-overlapping sensors; one in the front and one in the back and they both publish seen objects on separate topics. To monitor for objects seen by the UAV in total it is then useful to merge the content from both streams into one at runtime. The alternative would be to change in the source code so that the sensors also publish to a common topic although this could result in increased data traffic and merging with stream processing at runtime is a more dynamic and flexible solution since the merged stream can be created and destroyed dynamically at runtime.

3.5.3 State Synchronization

The synchronizer provided by the message_filters package in ROS is currently limited to nine incoming connections in C++. The rospy Python version supports more incoming connections but this comes at the prize of impaired performance due to the fundamental differences between Python and C++ which have been mentioned previously. Furthermore some of the synchronization policies would

be harder to implement due to the need to access the caches/buffers in which messages would be kept when using such a synchronizer. The existing synchronizer focuses on only sending synchronized states when new messages arrive rather than providing a synchronized stream of data which would be preferred for some DyKnow applications of this such as providing a steadily updated synchronized stream of states to the stream reasoner so it can evaluate temporal logic formulas.

For instance we might want to evaluate whether or not a car is faster than a UAV at any point in time during the next 30 seconds. In order to evaluate this a stream of states has to be provided to the reasoner so it can compare the speed of the UAV to the car. If the UAV has a higher speed value in any one of these states. The stream of synchronized states provides an abstraction of the continuous temporal domain by modeling it as a sequence of discrete states. In other words time is modeled as a series of points in time, all of which have a state containing the relevant information. The obvious downside is the inherent uncertainty. For instance in this example we might chose to have a sample rate of 900 ms and even if the UAV is faster in all of the sampled states we can't be entirely sure that the UAV was not faster for a very short period of time in between our samples. Even if the car was slower at 0 ms and 900 ms it might have been slightly faster at 500 ms when a state was not provided. Even with more states the sensors measuring the speeds have rates of their own as well as a margin of error for their measurements. A higher rate and better sensors would decrease the likelihood yet a certain degree of uncertainty is always unavoidable. Even if we use a very high sensor rate as well as sample rate it will still be a model of reality rather than a perfect description of reality itself yet what can be done is to chose the sample rates wisely to get accuracy which is sufficient to the task at hand and manageable for the reasoner. The differences between the values in the states could also be considered to estimate the relevant probabilities in these evaluations since it might be highly unlikely that the car could accelerate to the point it was faster given the difference in speed in all of the states.

Sync Example

A simple example of synchronization is when the reasoner needs a stream of states to evaluate if one autonomous UAV is faster or has a higher altitude than another at some point in time. The states need to be synchronized so the reasoner knows that the values in each state are comparable since they are regarding the same point in time. In this case we might have fields from four separate topics we need to synchronize: speed for uav1, speed for uav2, altitude for uav1 and altitude for uav2.

Synchronization and existing functionality in ROS

As mentioned in section 2.1.6 there is some support for synchronization in ROS. A few relevant downsides are that the current synchronization policy is not very well suited for some DyKnow related use cases, the Python version does not offer performance on par with compiled options and the use of templates limits the C++ version to 8 incoming streams where the different topic types have to be specified before runtime. The last limitation is important since in order to be able to synchronize streams at runtime in a dynamic way in line with the design requirements of the DyKnow framework there must then be readily compiled code for that specific combination of incoming message types and the combinatorics here dictate that the amount of compiled code will be huge in order to satisfy the number of combinations even for a low number of message types. This is much less of a problem if the incoming messages all share a common type though although there are still the other issues to take care of then.

Variadic templates might be a viable solution to increase the number of supported input streams although versions older than C++11 does not support this feature and as mentioned the amount of compiled code to cover even a fraction of the combinations available would be staggering which would make such a solution viable only when the type combinations which can occur at runtime are manageable. Since the focus here is on the system being dynamic and flexible at runtime the compromises needed to make such a template based solution work are not really acceptable. C++11 is not yet officially supported by ROS which makes this even more of a non-issue at the moment.

3.5.4 Basic Synchronization

A very basic version of state synchronization would be to wait until the maximum allow delays for each sample in the state has passed and then choose the samples closest to the synchronization time to be in that synchronized state.

3.5.5 Faster Synchronization

An improved synchronization algorithm which requires knowledge of when all relevant information has arrived has been designed by Heintz [15]. In general it works by determining when no more information relevant to the current synchronization time is due to arrive. When this is determined the synchronized state can be published. This requires knowledge of when all incoming messages relevant to the state are due to arrive. The benefit is that the states can be published faster at the expense of calculating when the next messages are due to arrive so whereas

the computational complexity is higher the delays can be lower.

3.6 The Stream Processing Language (SPL)

The purpose of the Stream Processing Language is to provide a readable and declarative way to create and use stream specifications since having to explicitly create all the messages is cumbersome and time-consuming. SPL, like many stream-based query languages, tries to make sure that the syntax looks relatively familiar to the user. In the case of SPL some of the syntax is inspired by SQL which hopefully aids readability.

The SPL language has been designed specifically for the purpose of creating Stream Specifications. Heintz has been invaluable in the design of the language and it is inspired in part by his previous work for DyKnow. The lack of native support for recursive structures in ROS is worth noting since that has influenced how the stream specifications look and therefore also influenced the syntax of SPL. The stream specifications were designed first and SPL was designed to build proper stream specifications rather than other way around.

3.6.1 Services

Having the SPL functionality in a separate service allows it to be an interface to the stream processor which can be used by both users and other nodes alike. The services provided by this service include returning the corresponding stream specification or creating the described stream by calling the stream processor directly.

Select SPL Example

The example explained in section 3.5.1 would correspond to the following line of SPL:

SELECT alt AS uav_1_altitude FROM dyknow_msgs/UAV:/info_about_uavs WHERE id = 1 WITH sample_period = 0.2

In this example alt is the name of the field containing the value of the altitude. The topic has the name info_about_uavs and contains messages of the type UAV from the package dyknow_msgs. Which UAV the message is about depends on

the field called id and the WHERE clause entails that only those messages where the id field value is equal to 1 are relevant, in other words we do not care about the altitudes of the other UAVs. The resulting stream will have a sample period of 0.2 seconds. The AS uav_1_altitude part is a renaming of the field alt to better describe the selected data when it is published on the stream.

Merge SPL Example

The example in section 3.5.2 would look like this using SPL syntax:

MERGE (SELECT * FROM dyknow_msgs/ObservedObjects:/front_sensor WHERE type = car, SELECT * FROM dyknow_msgs/ObservedObjects:/rear_sensor WHERE type = car) AS observed_car

Synchronization SPL Example

The SPL line which corresponds to the synchronization example in section 3.5.3 could look like this:

my_sync_stream_topic = SYNC (SELECT altitude FROM dyknow_msgs/Altitude:/altitude_ , SELECT speed FROM dyknow_msgs/Speed:/speed_1, SELECT altitude FROM dyknow_msgs/Altitude:/altitude_2 , SELECT speed FROM dyknow_msgs/Speed:/speed_2) WITH sample_period = 0.1

Backus Naur Form (BNF)

The design and formal syntax of SPL is defined below, the implementation follows it quite closely as discussed in the upcoming chapter.

```
NAMED_STREAM ::= NAMING_EXPR STREAM
STREAM ::= OPT-NAMING sync LP STREAMS RP with STREAM_CONSTRAINTS
 | merge LP STREAMS RP with STREAM_CONSTRAINTS
 | select SELECT_EXPRS from TOPIC where WHERE_EXPRS) with STREAM_CONSTRAINTS
STREAMS ::= STREAM | STREAM COMMA STREAMS
SELECT_EXPR ::= FIELD_ID (as PSTRING)?
SELECT_EXPRS ::= SELECT_EXPR | SELECT_EXPR COMMA SELECT_EXPRS
```

```
WHERE_EXPR ::= FIELD_ID EQ VALUE
WHERE_EXPRS ::= WHERE_EXPR | WHERE_EXPR and WHERE_EXPRS
FIELD_ID ::= STRING | STRING DOT FIELD_ID
TOPIC ::= TYPE COLON SLASH TOPIC_ID OPT-TOPIC_RATE
TOPIC_ID ::= STRING | STRING SLASH TOPIC_ID
PSTRING ::= STRING | STRING? PERCENT FIELD_ID PERCENT PSTRING?
STREAM_CONSTRAINTS ::= STREAM_CONSTRAINT |
                       STREAM_CONSTRAINT COMMA STREAM_CONSTRAINTS
STREAM_CONSTRAINT ::= MAX_DELAY_C | MAX_DEVIATION_C | START_TIME_C |
                      END_TIME_C | SAMPLE_PERIOD_C
```

Chapter 4

Implementation

In order to evaluate stream reasoning in ROS a node providing services related to stream processing was implemented. This module with its services is referred to as the stream processor. The stream processor is a ROS service node and therefore it responds to .srv requests. The services provided are related to the creation, destruction and management of streams. The streams to be created are described by stream specifications which are composed by several other smaller messages such as stream constraints, merge specifications and select specifications. Another node has been implemented to handle incoming SPL requests to parse them and create stream specifications based on their contents.

The requests to the stream processor service can for instance be sent by the stream reasoning coordinator and be based on the ontology for the UAVs in the system. Given the service requests the stream processor will then respond by subscribing to given topics in ROS and process incoming messages to for example provide the reasoner with a synchronized stream which the reasoner can use to evaluate its temporal and spatial logic formulas.

4.1 Stream Processing Operations

The operations the stream processor performs on incoming topics can be summarized by the following abstractions: select, rename, merge and synchronize. In summary: selecting relevant data, renaming it as needed, merging data when applicable and synchronizing all of the aforementioned into a state.

4.1.1 Select

A regular subscription in ROS gives you all of the information on that topic. Select is a way to specify exactly what feature to extract. For instance each message on the topic *uav_9* might contain the features altitude and speed in the fields called alt and spd. Irrelevant messages can also be discarded by specifying one or more identifying fields and their contents, as an example we might only want messages where the field called "id" is equal to 1 and thereby discarding messages published on that topic about other robots. Being able to select fields such as speed or altitude or both as wanted features is essential when creating the stream. Most of the select functionality is implemented as chains of filters which only allow relevant data to pass and report violations if the policies are broken.

4.1.2 Rename

If an AS clause has been used in the SPL or if renaming mappings have been specified in the stream specification through for instance semantic labels provided by the stream reasoning coordinator 3.2.2 the stream processor will use these associations in the fields rather than the old names.

4.1.3 Merge

Merge is implemented by having merged fields share the same buffer, i.e. merging them into the same buffer.

4.1.4 Synchronize

Each message in the resulting stream will be composed of the specified feature identifiers and their corresponding values synchronized. The synchronize procedures used are the faster synchronization procedures by Heintz [15] mentioned in the design chapter.

4.2 Stream Processor Services

Below is a list of the services provided by a stream processor node running in ROS, the messages which compose these services is described in detail in section 4.4:

Stream Processor Services:
`Create Stream From Spec`
`Get Stream`
`List Streams`
`Destroy Stream`

4.2.1 Create Stream From Spec

Create Stream From Spec is a service message where the request consists of a stream specification message which describes the entirety of the stream which the stream processor is requested to create. The stream processor will create this stream accordingly and the response will contain information indicating if it was successful, the topic of the resulting stream as well as a topic where possible violations to policies are reported. The content of the stream specification and the messages which it is composed of will be dealt with in more detail when the messages are explained.

contents of *CreateStreamFromSpec*.srv:

```
dyknow_msgs/StreamSpecification stream_spec
      string topic_name_of_published_stream
      dyknow_msgs/MergeSpecification[] merge_level_streams
            string merge_level_stream_name
            dyknow_msgs/SelectSpecification[] select_level_streams
                  string select_level_stream_name
                  string topic_name
                  string topic_msg_type
                  string topic_field
                  duration topic_sample_period
                  dyknow_msgs/FilterSpecification[] select_filters
                        bool constant
                        string object_identifier
                        string object_identifier_prefix
                        string field
                        string required_field_value
                        dyknow_msgs/StringPair[] renaming_map
                              string first
                              string second
      duration delay_before_approximation
      bool add_performance_timings_to_fields
      dyknow_msgs/StreamConstraints stream_constraints
            time start_time
            time end_time
```

```
            duration sample_period
            duration sample_period_deviation
            duration max_delay
            bool ordered_on_valid_time
            bool unique_valid_time
--
bool success
string error_message
string stream_topic
string stream_constraint_violation_topic
```

4.2.2 Get Stream

Get Stream provides a way to get the name of the stream_topic and stream_constraint_violatio
even after the response when it was created.

contents of *GetStream*.srv:

```
string stream_name
--
bool success
string error_message
string stream_topic
string stream_constraint_violation_topic
```

4.2.3 List Streams

List Streams is a convenient way to obtain information about all the streams generated by the Stream Processor node receiving this service call. When it is called at the terminal level through rosservice it prints such information.

contents of *ListStreams*.srv:

```
--
string[] stream_names
string[] stream_topics
string[] stream_constraint_violation_topics
```

4.2.4 Destroy Stream

As the name indicates Destroy Stream is used to destroy a stream, for instance because it is no longer needed.

contents of *DestroyStream*.srv:

```
string stream_name
--
bool success
string error_message
```

4.3 Stream Processing Language Related Services

SPL was implemented in a Python node which uses pyparsing to parse calls to the node. This keeps the design modular and leverages the parsing functionality which already exists in Python, thus there is no need for any dependencies on external libraries. The implementation of the language follows the design quite closely with the notable exception that the recursive definitions are flattened since ROS does not allow recursive structures in messages and SPL creates a Stream Specification which is in fact a rosmsg. The SPL services can create a stream specification from string or also use the stream specification to call the main stream processor node to start the stream immediately.

Create Stream From String

Create Stream From String can be used to create a stream from a SPL expression by parsing a string which is written in SPL to form a Stream Specification which is used to call CreateStreamFromSpec.

contents of *CreateStreamFromString*.srv:

```
string string_in_stream_processor_language_format
--
bool success
string error_message
```

```
string stream_topic
string stream_constraint_violation_topic
```

Create Stream Spec From String

Create Stream Spec From String can be used to create a Stream Specification from a SPL expression by parsing the incoming string.
contents of *CreateStreamSpecFromString*.srv:

```
string string_in_stream_processor_language_format
--
dyknow_msgs/StreamSpecification stream_spec
      string topic_name_of_published_stream
      dyknow_msgs/MergeSpecification[] merge_level_streams
            string merge_level_stream_name
            dyknow_msgs/SelectSpecification[] select_level_streams
                  string select_level_stream_name
                  string topic_name
                  string topic_msg_type
                  string topic_field
                  duration topic_sample_period
                  dyknow_msgs/FilterSpecification[] select_filters
                        bool constant
                        string object_identifier
                        string object_identifier_prefix
                        string field
                        string required_field_value
                        dyknow_msgs/StringPair[] renaming_map
                              string first
                              string second
      duration delay_before_approximation
      bool add_performance_timings_to_fields
      dyknow_msgs/StreamConstraints stream_constraints
            time start_time
            time end_time
            duration sample_period
            duration sample_period_deviation
            duration max_delay
            bool ordered_on_valid_time
            bool unique_valid_time
```

4.4 Messages

The stream specification is the message composing the request in a CreateStream-FromSpec service call. The stream specification is itself composed of other messages on different levels which all describe the stream which is to be created.

4.4.1 Stream Specification

The stream specification describes the entirety of the stream in its composing messages. It contains the name of the resulting stream as well as the name of the topic where violations to the polices will be published.

contents of *StreamSpecification*.msg:

```
string topic_name_of_published_stream
dyknow_msgs/MergeSpecification[] merge_level_streams
      string merge_level_stream_name
      dyknow_msgs/SelectSpecification[] select_level_streams
            string select_level_stream_name
            string topic_name
            string topic_msg_type
            string topic_field
            duration topic_sample_period
            dyknow_msgs/FilterSpecification[] select_filters
                  bool constant
                  string object_identifier
                  string object_identifier_prefix
                  string field
                  string required_field_value
                  dyknow_msgs/StringPair[] renaming_map
                        string first
                        string second
duration delay_before_approximation
bool add_performance_timings_to_fields
dyknow_msgs/StreamConstraints stream_constraints
      time start_time
      time end_time
      duration sample_period
      duration sample_period_deviation
      duration max_delay
      bool ordered_on_valid_time
      bool unique_valid_time
```

4.4.2 Merge Specification

The Merge Specification is a part of the Stream Specification. This part contains a set of Select Specifications, all of which are to be merged into one field in each resulting sample in the stream. The merge level stream name can be used in the naming of the merged field in the sample.
contents of *MergeSpecification*.msg:

```
string merge_level_stream_name
dyknow_msgs/SelectSpecification[] select_level_streams
     string select_level_stream_name
     string topic_name
     string topic_msg_type
     string topic_field
     duration topic_sample_period
     dyknow_msgs/FilterSpecification[] select_filters
          bool constant
          string object_identifier
          string object_identifier_prefix
          string field
          string required_field_value
          dyknow_msgs/StringPair[] renaming_map
               string first
               string second
```

4.4.3 Select Specification

The Select Specification contains the information needed to select specific data from a specific topic. It contains information about the topic such as its name and message type to make it easy to create a subscriber as well as the sample period to make it easier to predict when no more information relevant to our current sample will arrive. The Select Specification also contains a set of select filters to enforce policy based constraints on the subscriber side and report any violation to the constraints.
contents of *SelectSpecification*.msg:

```
string select_level_stream_name
string topic_name
string topic_msg_type
string topic_field
```

```
duration topic_sample_period
dyknow_msgs/FilterSpecification[] select_filters
      bool constant
      string object_identifier
      string object_identifier_prefix
      string field
      string required_field_value
      dyknow_msgs/StringPair[] renaming_map
            string first
            string second
```

4.4.4 Stream Constraints

The Stream Constraints specify a few more constraints. They are collected in a common substructure of the Stream Specification. The stream constraints include when the stream starts and ends although such functions are perhaps better handled by calling the stream processor service at the appropriate times. More importantly the structure contains the sample period of the stream which specifies how often a sample will be published. Max delay is the maximum allowed delay of the incoming messages. Sample period deviation is the deviation on each side of each regular sync time which is allowed. Support for a variable sync time is limited to stream start up at the moment although it is very possible to extend the support to better follow the design. The booleans ordered on valid time and unique valid time are on by default, mostly because if ordered on valid time is false there is the possibility that a more fitting value will arrive even though we got a message with a valid time which is more recent than the sync time, although the max delay would also be a factor in that case.
contents of *StreamConstraints*.msg:

```
time start_time
time end_time
duration sample_period
duration sample_period_deviation
duration max_delay
bool ordered_on_valid_time
bool unique_valid_time
```

4.4.5 Stream Constraint Violation

The Stream Constraint Violation message is used to report a violation to the constraints outlined in the messages above. It is published on a separate topic to offer the option to keep other nodes informed about these potential violations.
contents of *StreamConstraintViolation*.msg:

```
string stream_name
dyknow_msgs/StreamConstraints stream_constraints
      time start_time
      time end_time
      duration sample_period
      duration sample_period_deviation
      duration max_delay
      bool ordered_on_valid_time
      bool unique_valid_time
string constraint_violation_message
```

4.4.6 Sample

The stream processor has stream generators. Each stream generator creates a stream in the form of a topic which publishes Samples. Each sample in this stream can contain several fields which can be synchronized around a specific time.

The Sample structure is used for two things: 1) It is the elements composing the stream. In other words a stream is a topic which publishes messages of the message type dyknow_msgs::Sample.
2) Incoming messages from other topics which are to be processed are converted into Samples, these converted messages are processed by selecting relevant fields from them and taking fields which are to be synchronized to make up the samples mentioned above. contents of *Sample*.msg:

```
Header header
      uint32 seq
      time stamp
      string frame_id
time valid_time
dyknow_msgs/Field[] fields
      string type
      string name
      string value
```

4.4.7 Field

The composing unit of a Sample. There are very few primitive structures such as int, float and string which compose the more complex message types in ROS. Since there is no native support for recursive message structures the hierarchy of nested messages can be expressed by explicit paths through the submessages right down to the primitive ones. This conversion from message to a sample with fields is currently done through C++ conversion functions which are automatically generated by Python scripts. contents of *Field*.msg:

```
string type
string name
string value
```

4.5 Stream Specification Examples from SPL

SPL expressions can be used to generate a stream specification describing the stream. This stream specification can then be used to create the stream. Here follow a few examples illustrating the conversion of SPL expressions to stream specifications. The examples are continuations of the examples previously described in sections 3.5 and 3.6.

4.5.1 Select Example

SPL: SELECT alt AS uav_1_altitude FROM dyknow_msgs/UAV:/info_about_uavs WHERE id = 1 WITH sample_period = 0.2

Generated StreamSpec:

```
stream_spec:
     topic_name_of_published_stream:  dyknow_stream
     merge_level_streams:
          -
               merge_level_stream_name:  uav_1_altitude
               select_level_streams:
                    -
                         select_level_stream_name:  "
                         topic_name:  info_about_uavs
                         topic_msg_type:  dyknow_msgs/UAV
                         topic_field:  alt
                         topic_sample_period:
```

```
                              secs:  0
                              nsecs:  0
                         select_filters:
                              -
                                   constant:  False
                                   object_identifier:  "
                                   object_identifier_prefix:  "
                                   field:  id
                                   required_field_value:  1
                                   renaming_map:  []
    delay_before_approximation:
         secs:  0
         nsecs:  100000000
    add_performance_timings_to_fields:  False
    stream_constraints:
         start_time:
              secs:  0
              nsecs:  0
         end_time:
              secs:  0
              nsecs:  0
         sample_period:
              secs:  0
              nsecs:  200000000
         sample_period_deviation:
              secs:  0
              nsecs:  0
         max_delay:
              secs:  0
              nsecs:  100000000
         ordered_on_valid_time:  True
         unique_valid_time:  True
```

4.5.2 Merge Example

SPL: MERGE (SELECT * FROM dyknow_msgs/ObservedObjects:/front_sensor WHERE type = car, SELECT * FROM dyknow_msgs/ObservedObjects:/rear_sensor WHERE type = car) AS observed_car

Generated StreamSpec:

```
stream_spec:
     topic_name_of_published_stream:  dyknow_stream
```

```
merge_level_streams:
     -
          merge_level_stream_name:  observed_car
          select_level_streams:
               -
                    select_level_stream_name:  "
                    topic_name:  front_sensor
                    topic_msg_type:  dyknow_msgs/ObservedObjects
                    topic_field:  "
                    topic_sample_period:
                         secs:  0
                         nsecs:  0
                    select_filters:
                         -
                              constant:  False
                              object_identifier:  "
                              object_identifier_prefix:  "
                              field:  type
                              required_field_value:  car
                              renaming_map:  []
               -
                    select_level_stream_name:  "
                    topic_name:  rear_sensor
                    topic_msg_type:  dyknow_msgs/ObservedObjects
                    topic_field:  "
                    topic_sample_period:
                         secs:  0
                         nsecs:  0
                    select_filters:
                         -
                              constant:  False
                              object_identifier:  "
                              object_identifier_prefix:  "
                              field:  type
                              required_field_value:  car
                              renaming_map:  []
delay_before_approximation:
     secs:  0
     nsecs:  100000000
add_performance_timings_to_fields:  False
stream_constraints:
     start_time:
          secs:  0
          nsecs:  0
     end_time:
          secs:  0
```

```
            nsecs:  0
        sample_period:
            secs:  0
            nsecs:  100000000
        sample_period_deviation:
            secs:  0
            nsecs:  0
        max_delay:
            secs:  0
            nsecs:  100000000
        ordered_on_valid_time:  True
        unique_valid_time:  True
```

4.5.3 Synchronization Example

SPL: my_sync_stream_topic = SYNC (SELECT altitude FROM dyknow_msgs/Altitude:/al , SELECT speed FROM dyknow_msgs/Speed:/speed_1, SELECT altitude FROM dyknow_msgs/Altitude:/altitude_2 , SELECT speed FROM dyknow_msgs/Speed:/speed_2) WITH sample_period = 0.1

Generated StreamSpec:

```
stream_spec:
    topic_name_of_published_stream:  my_sync_stream_topic
    merge_level_streams:
        -
            merge_level_stream_name:  altitude
            select_level_streams:
                -
                    select_level_stream_name:  "
                    topic_name:  altitude_1
                    topic_msg_type:  dyknow_msgs/Altitude
                    topic_field:  altitude
                    topic_sample_period:
                        secs:  0
                        nsecs:  0
                    select_filters:  []
        -
            merge_level_stream_name:  speed
            select_level_streams:
                -
                    select_level_stream_name:  "
                    topic_name:  speed_1
```

```
                    topic_msg_type:  dyknow_msgs/Speed
                    topic_field:  speed
                    topic_sample_period:
                         secs:  0
                         nsecs:  0
                    select_filters:  []
          -
               merge_level_stream_name:  altitude
               select_level_streams:
                    -
                         select_level_stream_name:  "
                         topic_name:  altitude_2
                         topic_msg_type:  dyknow_msgs/Altitude
                         topic_field:  altitude
                         topic_sample_period:
                              secs:  0
                              nsecs:  0
                         select_filters:  []
          -
               merge_level_stream_name:  speed
               select_level_streams:
                    -
                         select_level_stream_name:  "
                         topic_name:  speed_2
                         topic_msg_type:  dyknow_msgs/Speed
                         topic_field:  speed
                         topic_sample_period:
                              secs:  0
                              nsecs:  0
                         select_filters:  []
     delay_before_approximation:
          secs:  0
          nsecs:  100000000
     add_performance_timings_to_fields:  False
     stream_constraints:
          start_time:
               secs:  0
               nsecs:  0
          end_time:
               secs:  0
               nsecs:  0
          sample_period:
               secs:  0
               nsecs:  100000000
          sample_period_deviation:
               secs:  0
```

```
      nsecs:  0
max_delay:
      secs:  0
      nsecs:  100000000
ordered_on_valid_time:  True
unique_valid_time:  True
```

4.6 Stream Processor Components

The stream processor has a stream generator for each stream it manages. Each stream generator in turn has one or more buffers for temporary storage of selected data from incoming messages which has passed a filter chain. The procedures in the stream generator puts together a stream of samples based on the contents of the buffers.

4.6.1 Stream Generators

A stream generator publishes a stream based on the stream specification. It is composed of several elements, among those are subscribers to the topics which can contain relevant data, filter chains to enforce policies, of one or more buffers which keep relevant data from the incoming topics. The stream generator implements the operations select, merge and sync to form a stream of states published on a specified new topic. It also implements most of the procedures for the aforementioned operations. Figure 4.1 shows the implementation of a stream generator. The buffers are updated when new data arrive. When it is likely that the newly arrived data has contributed something the synchronize procedure in the stream generator is called to see if a state for the current synchronization time can be published. The stream generator reacts to the incoming data to handle tasks related to forming streams such as the synchronization. The buffers contain the fields which the states published to the stream will contain. Merge is implemented by storing data which describe concepts similar enough to be deemed merged is sent to the same buffer.

4.6.2 Filter Chains

A stream generator has a filter chain for each subscriber to enforce policies. A filter can for instance deal with requirements such as having a maximum delay

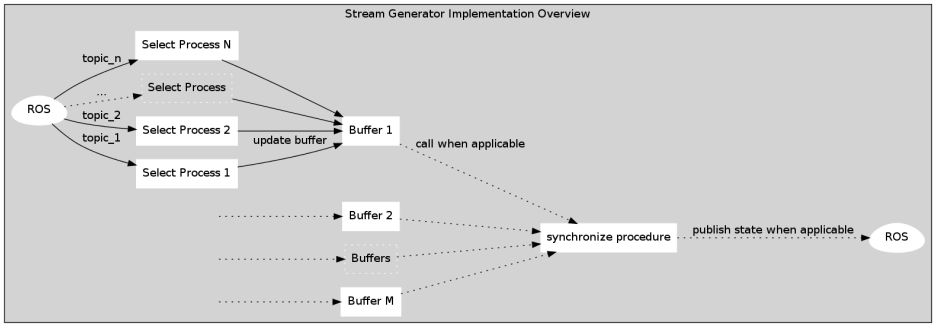


Figure 4.1: Implementation of a stream generator.

or disallowing out of order arrivals on topics. This is integral to providing the equivalent of policy-based subscribers in ROS.

4.6.3 Buffers

The buffers contain samples relevant to a field in the resulting state. For instance if two fields on two topics are to be synchronized in each state in the state stream, the stream generator will have two subscribers to these topics along with two buffers.

The buffers are all marked with a category which indicates the current condition with regard to synchronization, for instance the category can indicate if the buffer contains a sample which can be used as an approximation in the state for the current synchronization time and if a more recent approximation is due to arrive in time. Incoming messages from the topics are processed and if the content is deemed relevant a sample is put into the buffer. Since a single buffer in some cases can be affected by several different topics (in the instance of the merge operation) it also contains data with regard to when the next message for each of the topics relevant to the buffer is due to arrive.

4.7 Synchronization into States

The current implementation uses the synchronization algorithm in section 3.5.5 since it has been tried and offers advantages to simpler methods discussed in the analysis while still having a reasonable computational complexity in comparison to more complex alternatives. The procedures can be found in Heintz 2009 doctoral thesis in section 7.8.2[15].

If a start time is not explicitly mentioned the stream generator waits until it can first publish a complete sample and sets the first sync time according do that. The periodicity of the incoming messages often makes the following sync times quite suitable as well in terms of minimizing delays.

Expected Arrivals

When the new relevant data is expected to arrive is important since it is the deciding factor whether or not a state regarding the current sync time can be published now since no more relevant data will arrive or if we should wait for more data.

A map in the buffer keeps track of the topics affecting the contents and when the next message on each topic is due to arrive. Each pair in the map tells us the unique name of the topic as well as when the next message is due to arrive if we have some data about the topic's sample period. Defining the rate / sample period in advance makes sense since it is common in ROS and the framework also provides tools such as rostopic to figure the rate out at runtime. Furthermore the synchronization algorithm needs the sample periods to offer improvements in terms of earlier states even if it works fast and reliably otherwise as well.

Timeouts

The implementation mostly reacts to incoming service calls and messages although there is one notable exception: the timeouts. These are reached when publishing a state related to a specific sync time is overdue because of the max delay or delay before approximation. Timers to keep track of these timeouts is the obvious choice for the task but the documentation in ROS explicitly mentions that the Timers in ROS are not a real-time thread/kernel replacement and points out that they are useful mostly when there are not any real-time requirements. Timers were also tried in an earlier implementation with limited success yet they deserve a mention since they might become a more viable choice later on. Because of the issues with Timers the current implementation handles timeouts in two ways. One: letting the stream processor service node call the stream generators to check if a timeout has been reached and running the synchronization procedure if that is the case. Two: when new messages arrive to the stream generator and the synchronization procedure is run as a reaction to the incoming message callback the procedure checks for possible timeouts and deals with them accordingly. The system has to be fast enough to be able to process the messages and deal with timeouts so the publication time of the states in the stream do not deviate too much from the allowed delay before approximation. This is especially important in cases such as when feeding the reasoner with states to evaluate logic.

Chapter 5

Empirical Evaluation

The implementation of the design was evaluated through performance measurements investigating the performance impact on the system when, respectively; varying the number of stream generators, varying the number of merged fields and varying the number of fields from different topics to synchronize. By observing any system one unavoidably alters it to some extent, in this case the impact of the tests have been kept to a minimum by storing the performance timings in memory while running the tests and doing all the calculations afterwards.

The delays in the system can be divided groups.

First there is the delay between publishing on a topic and the subscriber receiving it. This delay is very much dependent on the spin rate in ROS and unavoidable when communicating on ROS topics.

There is also the internal delay in the stream processor between when a message arrives to it and the point in time where the message has been fully processed. Fully processed in this context means that the stream processor has gone through all the necessary procedures such as updating the buffers and, if possible, even publishing a state.

Lastly there is the so called perceived delay between the data's arrival to a stream generator and when it becomes part of a published state on a stream. This delay can be significantly higher since the stream processor sometimes has to wait a while to make sure that all the information for each synchronized state has arrived although it is important to note that it does not significantly exceed the maximum allow delay. If the stream generator can be sure that all the data regarding the specific synchronization time already has arrived there is no need to wait. The total delay will be higher than the perceived delay because of the aforementioned

transportation delays when communicating between nodes.

5.1 Performance measurements

Performance tests are vital to determine the overhead introduced by the DyKnow components. There are a lot of facets to the environment and focus with regard to performance measurements will be on latencies as well as throughput. The performance has been tested for varying numbers of stream generators, synchronized streams and merged streams, respectively. For each one of these tests three delays where measured. Average delays were measured as delay per field. 1) The transport delay between original publication on a topic to arrival at the stream processor, to show the overhead of ROS. 2) The internal delay between arrival at the stream processor and the point in time where the incoming message has been processed. 3) The perceived delay between original publication and publication on the state stream. It is worth noting that the delay between publication on the state stream and receiving the state is not included in the perceived delay so the actual delay should amount to the perceived delay plus a corresponding transport delay to compensate.

5.1.1 Measurement Overhead

To accurately measure the performance of the stream processing requires looking up and adding timing information which normally is not needed and this introduces some measurement overhead. Since IO operations are generally quite costly the overhead is minimized by eliminating them when running the tests and keeping track of only the accumulated delays and the average is calculated by dividing by the total number of instances.

Currently it is very hard to entirely eliminate the logging in ROS which makes it complicated to make the tests entirely reliable. Measures have been taken to minimize the logging done by ROS though although some logging such as the logging of subscribers and publishers in rosmaster.log was still active. An effort was made to try to avoid measuring at the same time as I/O operations were performed since debugging printouts reliably impaired performance.

Each specific performance test needs a varying number of active topics to run. This introduces the question whether or not only the required topics should be running. Having a high quantity of topics running at once each of which can also have a high rate of messages introduces a load on the system which can affect the tests. In order to make the performance measurements comparable all of the topics used in a performance measurement should be running even if they are not

used in that specific test since they will be used in later tests and they should have the same overhead in terms of background processes.

5.1.2 Settings and Test System

Callbacks in ROS are processed in a node by calling spin at a predefined rate. These tests were run without a rate defining a waiting period between the spins since waiting between each spin could have an impact on the test results. As a result the CPU utilization during these tests were high since the system was constantly checking for callbacks to process, as an example simply having an empty node calling ros::spin causes similar load on the system which indicates that this is independent of the stream processing implementation. CPU load is of course much more reasonable at a fixed rate for ros::spin and the faster the stream processing is the faster rate can be used. While performing these tests the publishers had a fixed rate between spins and when there were subscribers to the streams from the stream generators the subscribers were run without waiting period between the spins in order to receive the messages from the stream as fast as possible.

The system used ran ROS Electric Emys on the operating system Lubuntu 10.10 (www.lubuntu.net, in essence a lightweight Ubuntu 10.10 with LXDE) on an older laptop: Dell Inspiron 640m with an Intel Core 2 T5500 @ 1.66 Ghz, 2 GB RAM and upgraded with an Crucial m4 SSD running firmware 0309. System specifications are listed so the results can be reproduced and compared, in practice the stream processing functionality described in this thesis should be able to run on systems capable of running ROS itself since the stream processor module does not use any external libraries beyond what is included in ROS. This includes embedded systems such as the autonomous UAVs discussed in this thesis.

A 120 publishers each publishing at a rate of 20 messages per second were set up along 120 subscribers to the streams (not all of them published to). The message type for these topics was UAV which was described in section 2.1.4. Through services the test system starts and destroys the required streams by formulating SPL expressions. The SPL expressions all had a sample rate of 0.2 (one sample every 200 milliseconds). When varying a variable in the tests the stream(s) were always set up to subscribe to a large set of topics, for instance when setting up a hundred stream generators they all subscribed to different topics and when synchronizing a hundred fields they all came from different topics from the pool of publishers running.

5.1.3 Internal Latencies and Perceived Latencies

The latencies are divided into two categories. The first category of latencies is called internal latencies and refers to the delay between a subscriber in the stream processor receiving the message and when it is done processing it. The perceived delay on the other hand includes more than just this processing such the delay between publication and arrival and delays caused by having to wait for messages to arrive on others topics to give us an approximation of a particular feature for a certain sync time. The perceived latencies measures the total overhead as viewed by the client whereas the internal latencies are more closely related to the efficiency of the implementation.

5.1.4 Varying the Number of Stream Generators

These charts show how the performance of the system is affected by having multiple stream generators running in a stream processor. Each stream generator is synchronizing fields from two topics.

Internal Delays When Varying the Number of Stream Generators

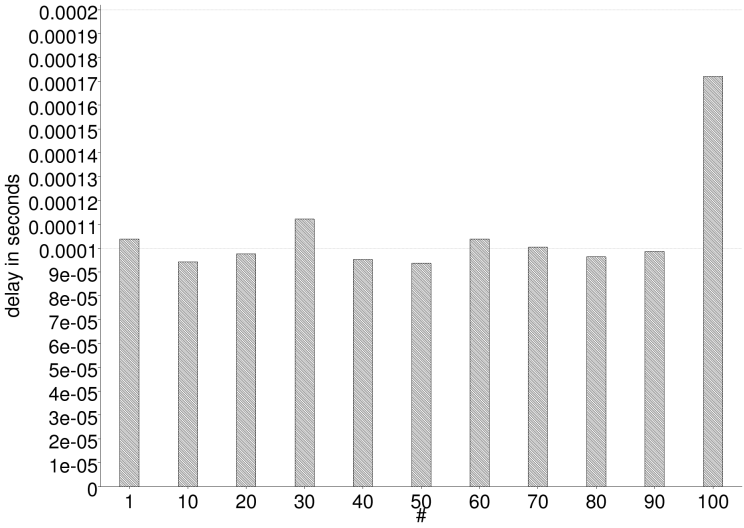
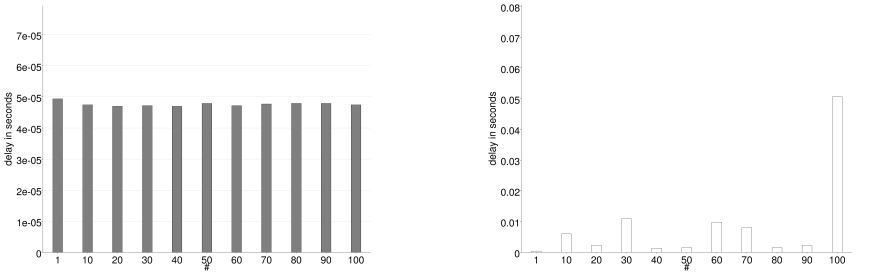


Figure 5.1: The average internal delays per field for a varying number of stream generators running.



(a) stream generator min internal

(b) stream generator max internal

Figure 5.2: Stream Generator min/max internal delays

Perceived Delays When Varying the Number of Stream Generators

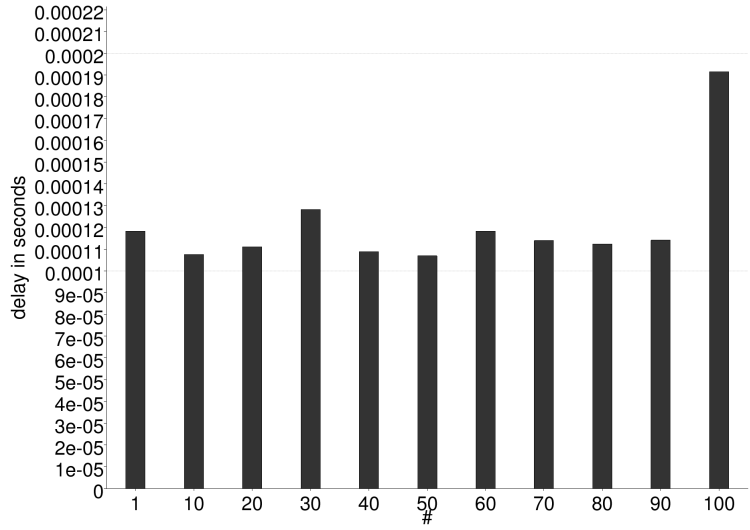
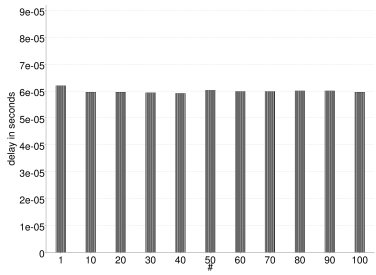
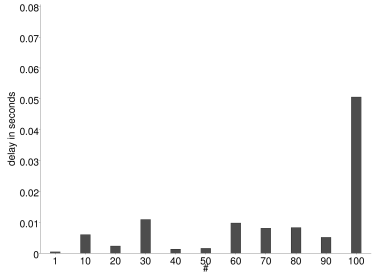


Figure 5.3: The average perceived delays per field for a varying number of stream generators running.



(a) stream generator min perceived



(b) stream generator max perceived

Figure 5.4: Stream Generator min/max perceived delays

Transport Delays When Varying the Number of Stream Generators

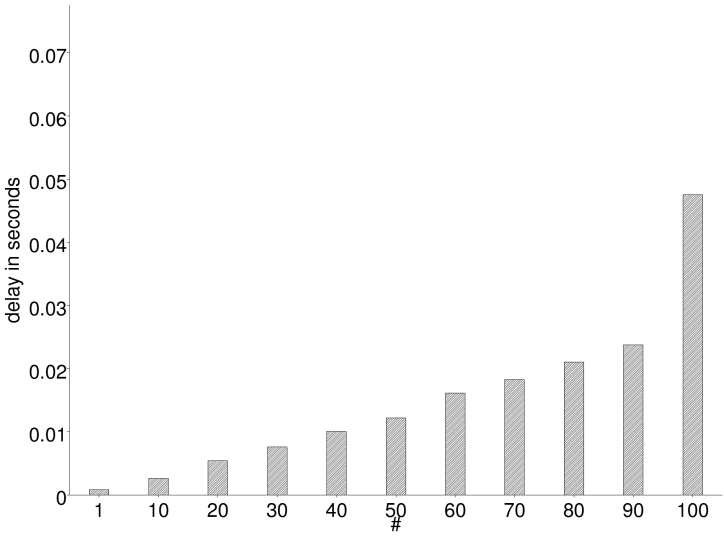
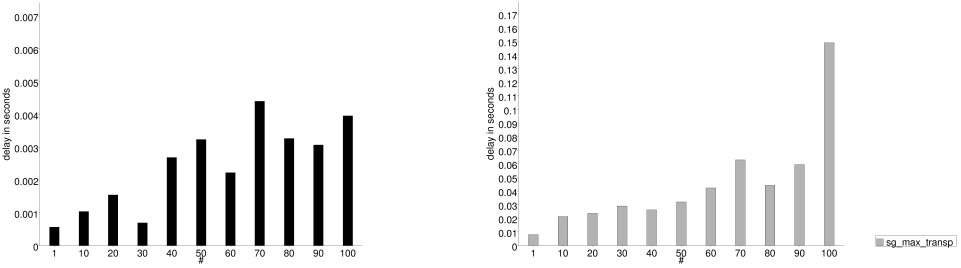


Figure 5.5: The average transport delays per field for a varying number of stream generators running.



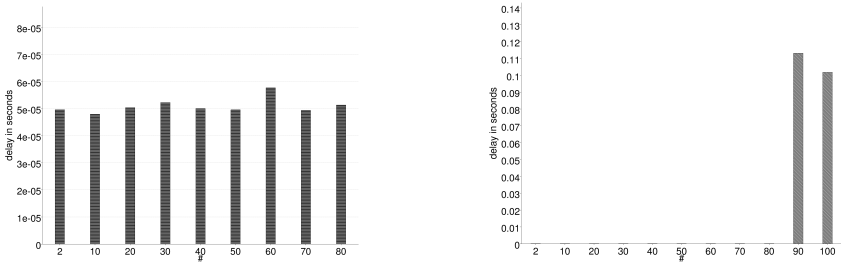
(a) stream generator min transport (b) stream generator max transport

Figure 5.6: Stream Generator min/max transport delays

5.1.5 Varying the Number of Select Statements to Merge

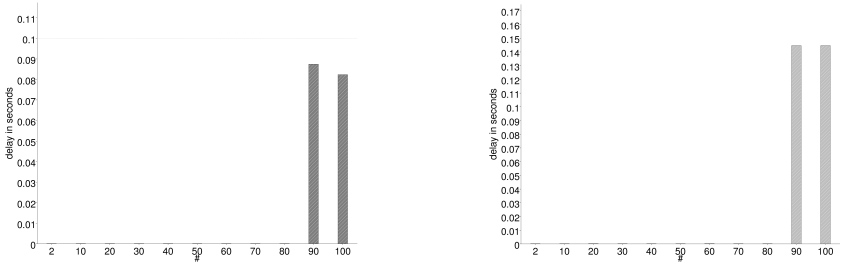
These charts show how the performance of the system is affected when merging from several topics in the stream processor.

Internal Delays When Varying the Number of Select Statements to Merge



(a) 2-80 (b) 2-100

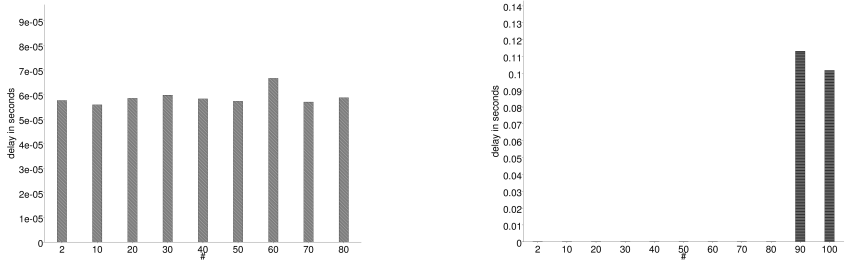
Figure 5.7: The average internal delays per field for a varying number of topics to merge from.



(a) merge min internal (b) merge max internal

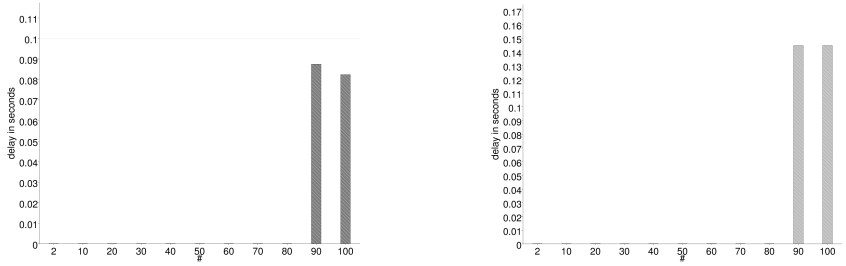
Figure 5.8: Merge min/max internal delays

Perceived Delays When Varying the Number of Select Statements to Merge



(a) 2-80 (b) 2-100

Figure 5.9: The average perceived delays per field for a varying number of topics to merge from.



(a) merge min perceived (b) merge max perceived

Figure 5.10: Merge min/max perceived delays

Transport Delays When Varying the Number of Select Statements to Merge

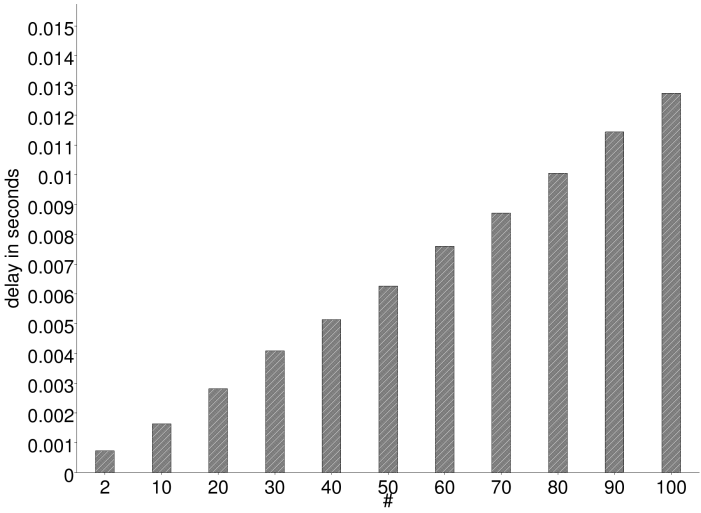
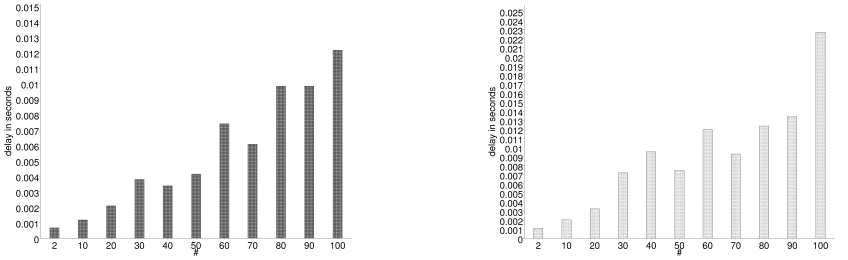


Figure 5.11: The average transport delays per field for a varying number of topics to merge from.



(a) merge min transport

(b) merge max transport

Figure 5.12: Merge min/max transport delays

5.1.6 Varying the Number of Select Statements to Synchronize

These charts show how the performance of the system is affected when synchronizing from several topics in the stream processor.

Internal Delays When Varying the Number of Select Statements to Synchronize

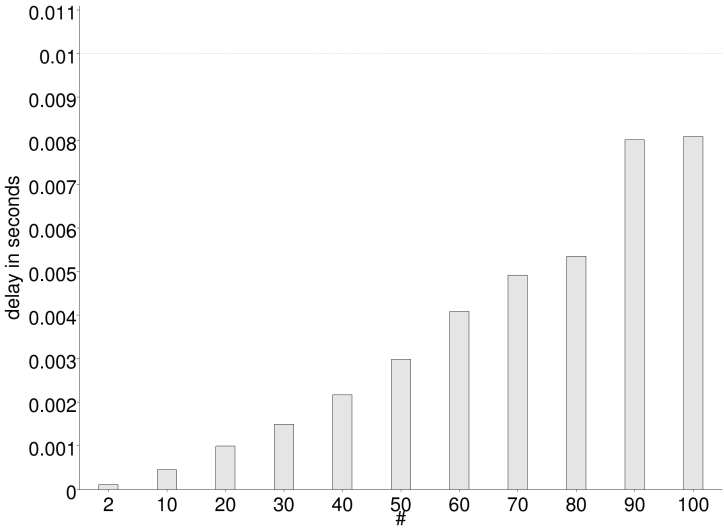
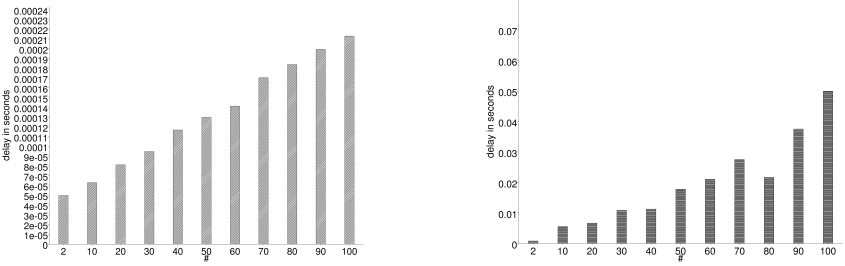


Figure 5.13: The average internal delays per field varying number of topics to synchronize.



(a) sync min internal

(b) sync max internal

Figure 5.14: Sync min/max internal delays

Perceived Delays When Varying the Number of Select Statements to Synchronize

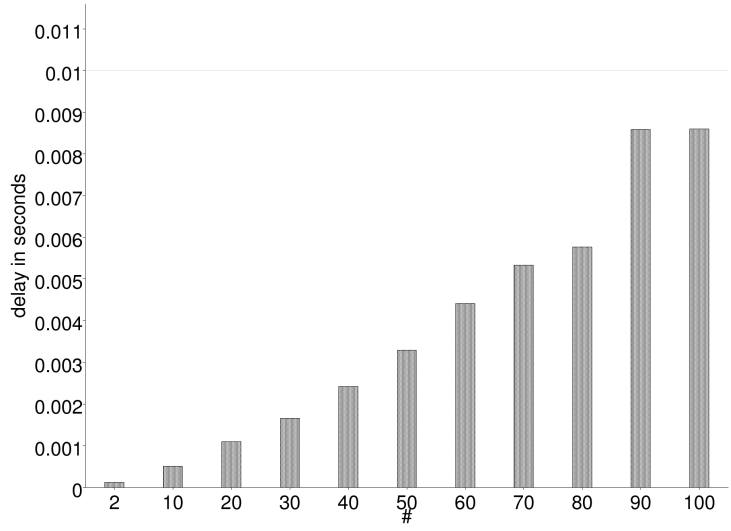
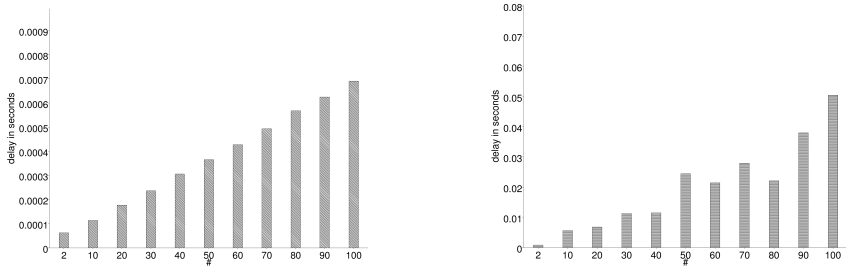


Figure 5.15: The average perceived delays per field for a varying number of topics to synchronize.



(a) sync min perceived

(b) sync max perceived

Figure 5.16: Sync min/max perceived delays

Transport Delays When Varying the Number of Select Statements to Synchronize

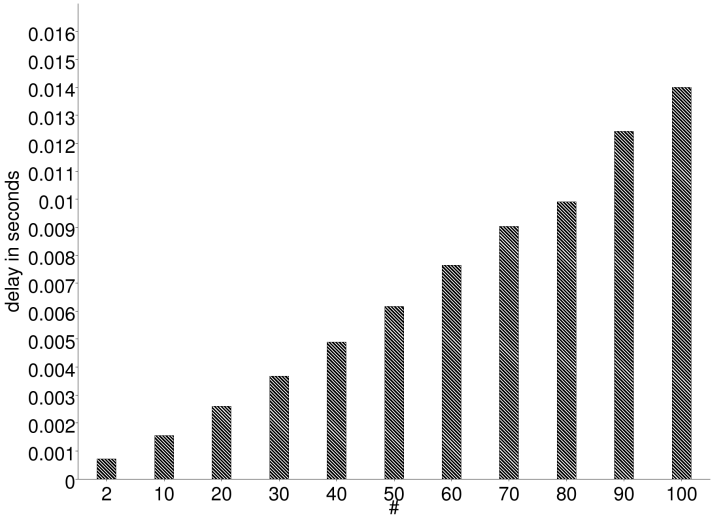
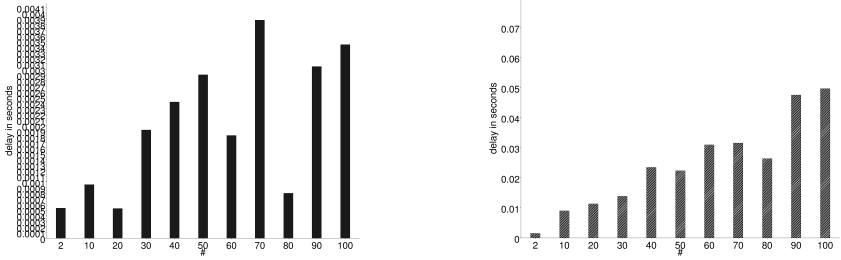


Figure 5.17: The average transport delays per field for a varying number of topics to synchronize.



(a) sync min transport (b) sync max transport

Figure 5.18: Sync min/max transport delays

5.2 Discussion

5.2.1 Latencies

Transport Delays

With constantly spinning nodes the delays for communicating between nodes is negligible when there are few messages being sent. Introducing a delay between each spin for subscribing nodes also introduces a corresponding delay since the node has to wait until it checks for updates on the topic. Having constantly spinning nodes to minimize delays can be undesirable because of the extra processing power required for the exuberant spinning. Even at very high rates the processing usage is very manageable compared to forgoing the delay between each spin entirely.

When sending hundreds of messages at once the transport delays seem to increase by quite a lot though. Anomalous values in the performance charts which occur at the higher numbers in the tests correlate quite well with the transport delays. Separate measurements where only the transport delays are measured involving only regular ROS publishers and subscribers seem to confirm this. Probably since the messages are published in batches at roughly the same time which causes some congestion with regard to the throughput since incoming and outgoing messages are processed one by one in the callbacks when the nodes are spinning. The transport delays are not that big of an issue for the stream processor up until a point. This could be where the accumulated transport delays exceed the max delay allowed by the policies in the stream generators. At this point it is a problem since the predictions for when the messages are supposed to arrive do not reflect the actual arrival and it becomes hard for the stream processor to receive the messages with the data relevant for each sample. A lower rate eliminates this issue of congestion entirely and produces much nicer graphs as can be seen. Discussing current limitations of a system is always more interesting though.

Perceived Delays

Another thing to notice is that the perceived delays can also depend on the timed events in the form of timeouts when states should be published. The management of such timed events was discussed in 4.7 and their accuracy depend on the system being able to deal with incoming messages fast enough and check whether the timed events have been reached. This matters since handling these events in an expedient manner is what enables the states in the stream to get published in a timely fashion. The ROS real-time package rosrt was not used in the current implementation since it is experimental and not considered API-stable right

now yet this might be a viable option in the future for timing critical publishers/-subscribers and timed events which are designed for real-time applications. The currently available functionality should be sufficient for most purposes though considering the performance.

In these tests both the internal and the perceived delays were generally very low. Much of this is thanks to how the stream processor chooses the first synchronization time based on the first incoming messages as described earlier and it is important to note that the perceived delays can be much more significant if the sync times are inconvenient with regard to that they often require the stream generators to wait for incoming messages. In these tests the stream generators did not have to wait long for the messages to arrive which makes the perceived delays quite close to the internal delays. It is also noteworthy that the perceived delays do not include the transport delays which can be significant when the system has to deal with a large amount of streams.

Perceived delays can be amplified by factors such as if the nodes seldom spin to deal with callbacks or if the sync times are constantly out of sync with when the incoming messages are actually published. Implementing an algorithm which take the sample period deviation into consideration might improve that latter aspect. If an explicit start time for the stream is omitted the stream generator seems to do a fairly good job of setting the first sync time to make this much less of an issue though as seen in the performance tests.

Publishing the incoming messages at roughly the same times and starting the stream generators without deliberately choosing dispersed sync times might have a negative impact on the performance in these tests as a more even distribution temporally would ease the load when synchronizing. Starting several streams at once seems like a common use case though considering applications such as formula evaluation and the performance under these conditions is encouraging.

Internal Delays

As can be expected the internal delays are always lower than the perceived delays even though they are quite similar here. It is worth noting that both the perceived and internal delays are much lower than the transport delays by quite a large factor.

Dealing with Delays

Lowering the rates on the incoming topics and the outgoing stream is one way to solve the congestion issue. Figure 5.19, 5.20 and 5.21 illustrate what happens when running publishers on 200 topis and creating resulting streams with a sample

period of 400 ms where each incoming topic has a period of 100 ms. To avoid congestion the rates only really need to be lowered when the tests reach very high numbers on the x-axis and they could probably be higher overall without congestion being an issue. For the lesser numbers on the x-axis the rates can be vastly higher. A good solution to this which is in line with the overarching design goals of DyKnow would be to dynamically adapt the rates in the system at run-time to avoid congestion while retaining higher rates when possible.

As demonstrated by the figures; increasing the sample period gives the system more time to deal with the transportation delays and thus preventing congestion. As expected the synchronization is slightly more demanding than the other operations and the computational complexities for the tests were encouraging when congestion was averted. With regard to the tests varying the number of stream generators and varying the number of selects per merge; they both produced a constant delay per field.

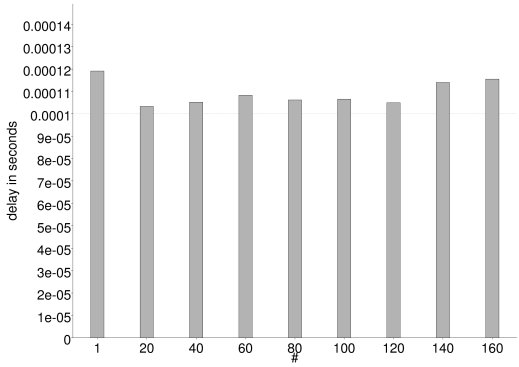


Figure 5.19: The average perceived delays per field for a varying number of stream generators.

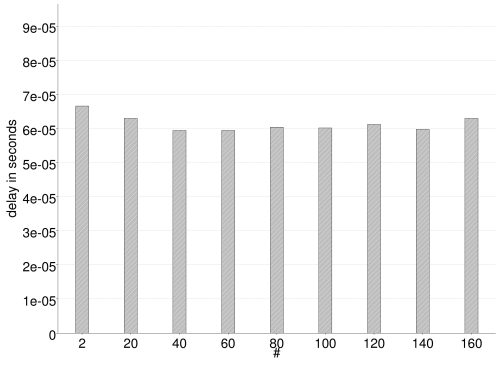


Figure 5.20: The average perceived delays per field for a varying number of topics to merge.

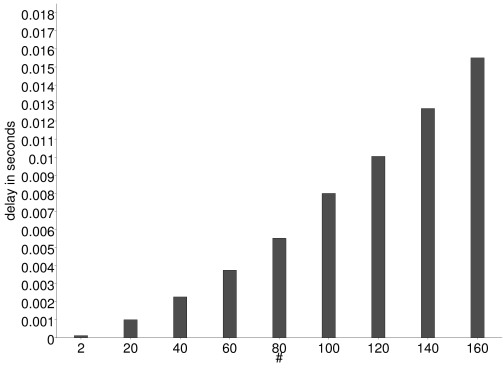


Figure 5.21: The average perceived delays per field for a varying number of topics to synchronize.

Throughput

In general in ROS a too high throughput can result in dropped messages on the topics. Dropped messages could result in the progressor failing to receive vital information about a formula it is evaluating. Although monitoring if we drop messages is easily checked by reading the message header this is desirable to avoid altogether by keeping the rates reasonable enough for ROS to handle.

Concerning stream reasoning throughput in ROS there are certain theoretical limitations such as when either the stream processing can't be done fast enough in the stream processor node to provide the stream or some of the topics have an overwhelming load which causes messages to be dropped. The frequencies of input (topics to synchronize) as well as output (sample stream to progressor) are relevant. One way to deal with dropped messages as well as packetloss in general would be to use the sequence ID in cached message headers to see if any messages are missing. However, whether or not this would be useful is arguable, especially since congestion could be a larger issue as shown by the increasing transport delays.

For very high sample periods it could be desirable to integrate the progressor in the stream processor node to entirely eliminate the risk of dropped message between them although one of the benefits of having different nodes is that you can distribute them on different cores or platforms and balance the load. Furthermore rates high enough to cause dropped messages are probably undesirable for the state streams since the progressor has to handle the incoming data which is why it is good to specify a reasonable sample rate in the stream specification even if the stream processor is fast enough to deliver a higher rate. Therefore it should not be a problem for the stream processing in practice.

5.2.2 Testing Overhead

Regarding the overhead introduced by the performance tests: measurement tools such as valgrind and sysprof seem to indicate the function ros::Time::now() does introduce a small amount of overhead since this function is called far more frequently to add performance timings.

5.2.3 Scalability

The current implementation seems to be scalable in large part due to how fast it is able to process incoming and outgoing data sequentially which enables the server node to process a large quantity of streams without issues. Further testing with several nodelets instead of a stream processor node could however be interesting in terms of scalability and micromanagement of thread handling could improve performance when running a large amount of streams at once. The option to easily disable logging globally in ROS would also help since I/O operations are costly and one does not want to fill the available space on an embedded system over time.

DyKnow and ROS both have attributes which makes the stream processing very scalable. ROS' fundamental architecture of nodes connected by topics makes it quite scalable since it is then easy to distribute processing power by have the nodes run on different platforms dealing with different degrees of complexity. Doing all the computations on the embedded system inside the robot itself isn't always desirable if battery life can be extended by doing calculations elsewhere, such as locally in another system or in the cloud. The modularity of ROS' node based architecture enables efficient distributed computing for instance in the form of cloud computing and multicore computing[17]. Another aspect of DyKnow worthy of mention here is dynamic reconfiguration based on runtime data which could play an integral part of load balancing the streams used in ROS to further improve the scalability.

Chapter 6

Conclusions and Future Work

The analysis of the underlying design of ROS support the notion that ROS is very much extensible with DyKnow inspired functionalities such as stream reasoning and the implementation outlined here corroborates that while offering good performance with a small overhead.

Extending ROS with such functionality was made easier by the similarities between concepts in ROS and DyKnow although some hurdles were overcome in the process as well. The stream reasoning architecture is designed with these similarities and the concept of modularity in mind. Stream processing operations to select, merge and synchronize streams of data in ROS work well on their own and offer potential in many other applications such as in the aforementioned stream reasoning architecture.

The SPL syntax was thought of after getting most if not all stream processing functionality to work. If one started with the SPL there is a good chance the syntax would have been less complex in terms of size. There would probably be some downsides with that approach, for instance that the BNF would probably contain some relations which are difficult to implement due to lack of recursive messages in ROS and also it would be another layer of abstraction which could change a lot from iteration from iteration during the development process. Therefore developing the Stream Specification and how it is handled first made more sense. Having SPL as a direct interface with the stream processor and omitting the step of having a stream specification entirely would however be an alternative which would allow an arbitrary C++ structure to describe the streams, this however might be less modular and makes the parsing much more important.

The development process has been very iterative with lots of different implementations along the way to try out new things and lots of reflection whether or not they are easy to use and efficient. Ease of use and legibility are however abstract concepts which are hard to measure objectively without extensive user studies yet this must not be used as an excuse to ignore their importance and they have certainly at least been taken into account here. Admittedly the starting point has been to develop solutions which work efficiently and then in each iteration trying to add, replace and improve. Sometimes different ways of doing things such as for instance policy-based subscriptions have been implemented in parallel to compare the solutions. There are a lot of aspects to consider while writing code and trying to achieve both elegance and speed and seeing solutions side by side can help immensely at times.

6.1 Future Work

There are several directions in which this work can be explored further. For instance the system could adapt to take variables such as the system's current free resources into account when deciding on things like the publishers and the streams sample rates. Furthermore, as discussed in section 5.2.1 the system would benefit from the feature to dynamically change these rates at runtime to ensure smooth performance even under extraordinary circumstances. Such load balancing related work would have the added benefit of being applicable in several of the many domains where streams of data are involved.

The sample period deviation could use some further investigations as to how a solution such as the one presented in the analysis would affect aspects of the system's performance, especially with regard to the perceived delay when synchronizing a multitude of incoming asynchronous topics. The stream processor could also be extended to provide higher level stream operations, such as logic, arithmetic and complex event processing. Naturally SPL can then also be extended to reflect these changes. Aside from that the DyKnow stream reasoning architecture in ROS can also be extended with more modules and be developed further with for instance better support for distributed systems with heterogeneous robotics platforms. The work on ROS also continues and as always more research is required for robots in general and autonomous UAVs in particular to reach their full potential to help us in the future, no matter if it is with regard to helping us with mundane chores or helping us with search and rescue missions where lives are at stake.

Bibliography

[1] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB JournalâThe International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[4] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Incremental reasoning on streams and rich background knowledge. *The Semantic Web: Research and Applications*, pages 1–15, 2010.

[5] D.F. Barbieri, D. Braga, S. Ceri, E.D. Valle, and M. Grossniklaus. Querying RDF streams with C-SPARQL. *ACM SIGMOD Record*, 39(1):20–26, 2010.

[6] M. Beetz, L. MoÌsenlechner, and M. Tenorth. CRAMâA Cognitive Robot Abstract Machine for everyday manipulation in human environments. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1012–1017. IEEE, 2010.

[7] A. Bifet and E. Frank. Sentiment knowledge discovery in twitter streaming data. In *Discovery Science*, pages 1–15. Springer, 2010.

[8] L. Cánovas, R. Cañavate, and A. Marín. On the convergence of the weiszfeld algorithm. *Mathematical programming*, 93(2):327–330, 2002.

[9] E. Della Valle, S. Ceri, D. Braga, I. Celino, D. Frensel, F. van Harmelen, and G. Unel. Research chapters in the area of stream reasoning. In *Proc. of the 1st International Workshop on Stream Reasoning (SR2009), CEUR Workshop Proceedings*, volume 466, pages 1–9, 2009.

[10] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *Intelligent Systems, IEEE*, 24(6):83–89, 2009.

[11] Z. Dragisic. Semantic matching for stream reasoning. Master's thesis, Linköpings universitet, 2011.

[12] F. Heintz, J. Kvarnström, and P. Doherty. Stream Reasoning in DyKnow: A Knowledge Processing Middleware System. In *Proc. 1st International Workshop on Stream Reasoning*. Citeseer. In Stream Reasoning Workshop, Heraklion, Crete.

[13] F. Heintz, J. Kvarnström, and P. Doherty. Bridging the sense-reasoning gap: DyKnow-Stream-based middleware for knowledge processing. *Advanced Engineering Informatics*, 24(1):14–26, 2010.

[14] F. Heintz, J. Kvarnström, and P. Doherty. Stream-Based Reasoning Support for Autonomous Systems. In *Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 183–188. IOS Press, 2010.

[15] Fredrik Heintz. *DyKnow: A Stream-Based Knowledge Processing Middleware Framework*. PhD thesis, Linköpings universitet, 2009.

[16] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming sql standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, 2008.

[17] D Kohler, R. Hickman, K Conley, and B Gerkey. Google I/O 2011: Cloud Robotics. 2011.

[18] D. Lazarovski. Extending the stream reasoning in dyknow with spatial reasoning in rcc-8. Master's thesis, Linköpings universitet, 2012.

[19] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[20] P. Thati and G. Rosu. Monitoring algorithms for metric temporal logic spec-

ifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.