

TECHNICAL UNIVERSITY OF CRETE, GREECE
DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING

Narukom

A Distributed, Cross-Platform, Transparent Communication Framework for Robotic Teams



Evangelos E. Vazaios

Thesis Committee

Assistant Professor Michail G. Lagoudakis (ECE)

Assistant Professor Vasilios Samoladas (ECE)

Assistant Professor Nikolaos Vlassis (DPEM)

Chania, February 2010

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Narukom

Ένα Κατανεμημένο, Διαλειτουργικό, Διάφανο
Πλαίσιο Επικοινωνίας για Ρομποτικές Ομάδες



Ευάγγελος Ε. Βαζαίος

Εξεταστική Επιτροπή

Επίκουρος Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΗΜΜΥ)

Επίκουρος Καθηγητής Βασίλειος Σαμολαδάς (ΗΜΜΥ)

Επίκουρος Καθηγητής Νικόλαος Βλάσσης (ΜΠΔ)

Χανιά, Φεβρουάριος 2010

Abstract

Teams participating in a game require means of interaction among their members in order to coordinate their efforts and achieve a common goal. For human teams this is a natural skill and comes without much effort, however for robotic teams communication is not trivial. This thesis describes a distributed communication framework for robotic teams developed for the Standard Platform League (SPL) of RoboCup, the annual international robotic soccer competition. During a game, robot players need to share perceptual, strategic, and other team-related information with their team-mates typically over the wireless network. Maintaining a synchronized copy of each robot's data on each node of the network is both inefficient and unrealistic for real-time systems. Our proposal suggests a distributed and transparent communication framework, called Narukom, whereby any node (robot or remote computer) can access on demand any data available on some other node of the network in a natural way. The framework is based on the publish/subscribe paradigm and provides maximal decoupling not only between nodes, but also between threads on the same node. The data shared between the nodes of the team are stored on local blackboards which are transparently accessible from all nodes. To address synchronization needs, which are common in robotic teams, we have integrated temporal information into the meta-data of the underlying messages exchanged over the network. Narukom's distributed nature and platform independence make it an ideal base for the development of coordination strategies and for distribution of resource-intensive computations over different nodes. Narukom has been created for and has been successfully used by our RoboCup team, Kouretes, for exchanging data between the Nao robots of the team, however it can be used in any distributed network application with similar communication needs.

Περίληψη

Κάθε ομάδα που συμμετέχει σε ένα παιχνίδι χρειάζεται κάποιο τρόπο επικοινωνίας μεταξύ των μελών της προκειμένου αυτά να συντονίσουν τις προσπάθειές τους και να πετύχουν τον κοινό στόχο τους. Για ανθρώπινες ομάδες, η επικοινωνία είναι τελείως φυσική διαδικασία και αναπτύσσεται χωρίς ιδιαίτερο κόπο, ωστόσο για ρομποτικές ομάδες είναι ένα μη τετριμμένο θέμα. Η παρούσα διπλωματική εργασία περιγράφει ένα καταναμημένο πλαίσιο επικοινωνίας για ρομποτικές ομάδες που αναπτύχθηκε για το πρωτάθλημα Standard Platform League του RoboCup, του ετήσιου διαγωνισμού ρομποτικού ποδοσφαίρου. Κατά τη διάρκεια ενός αγώνα, οι ρομποτικοί παίκτες χρειάζονται να μοιράζονται πληροφορίες που σχετίζονται με την αντίληψή τους, τη στρατηγική τους, ή άλλα στοιχεία της ομάδας με τους συμπαίκτες τους, κατά κανόνα μέσω του ασύρματου δικτύου. Η διατήρηση ενός πάντα ενημερωμένου αντιγράφου των δεδομένων του κάθε ρομπότ σε κάθε έναν από τους υπόλοιπους κόμβους του δικτύου είναι αναποτελεσματική και μη ρεαλιστική προσέγγιση, ειδικά για τέτοια συστήματα πραγματικού χρόνου. Η πρότασή μας συνιστά ένα εύχρηστο, καταναμημένο και διάφανο πλαίσιο επικοινωνίας, με την επωνυμία Narukom, μέσω του οποίου κάθε κόμβος του δικτύου (ρομπότ ή απομακρυσμένος υπολογιστής) έχει τη δυνατότητα να προσπελάσει όλα τα διαθέσιμα δεδομένα σε κάποιον άλλο κόμβο του δικτύου με απλό και φυσικό τρόπο. Το πλαίσιο βασίζεται στο μοντέλο publish/subscribe, το οποίο προσφέρει τη μεγαλύτερη δυνατή ανεξαρτησία μεταξύ των κόμβων του δικτύου, αλλά και μεταξύ των νημάτων που τρέχουν σε κάθε κόμβο. Τα δεδομένα που μοιράζονται μεταξύ τους οι κόμβοι αποθηκεύονται σε τοπικές δομές (blackboards) που είναι διάφανα προσβάσιμες από τους υπόλοιπους κόμβους. Επιπλέον, για να αντιμετωπισθεί η ανάγκη συγχρονισμού των δεδομένων, που είναι συνήθης σε ρομποτικές ομάδες, προθέσαμε χρονική πληροφορία στα μετα-δεδομένα των μηνυμάτων που διακινούνται στο δίκτυο. Η καταναμημένη αρχιτεκτονική του Narukom και η ανεξαρτησία του ως προς την πλατφόρμα εκτέλεσης το καθιστούν ιδανική βάση για την ανάπτυξη περίπλοκων στρατηγικών συντονισμού και για την κατανομή απαιτητικών υπολογισμών σε διαφορετικούς κόμβους. Το Narukom δημιουργήθηκε και χρησιμοποιήθηκε με επιτυχία για την ανταλλαγή δεδομένων μεταξύ των ρομπότ Nao της ομάδας Κουρήτες του Πολυτεχνείου Κρήτης, ωστόσο μπορεί να χρησιμοποιηθεί και σε άλλες καταναμημένες εφαρμογές με παρόμοιες ανάγκες επικοινωνίας.

Contents

1	Introduction	1
1.1	Thesis Contribution	2
1.2	Thesis Outline	3
2	Background	5
2.1	RoboCup Competition	5
2.1.1	The Standard Platform League	6
2.1.2	Simulation League	7
2.1.3	Small Size League	8
2.1.4	Middle Size League	8
2.1.5	Humanoid League	9
2.2	Team Kouretes	10
2.3	The Aldebaran Nao Robot	11
2.3.1	Sensors and Actuators	12
2.3.2	NaoQi	13
2.4	Communication Technology	13
2.4.1	UDP	13
2.4.2	The Publish/Subscribe Paradigm	14
2.4.3	Google Protocol Buffers	14
2.4.4	Blackboard	15
2.4.5	Monitor	16
2.4.6	Boost C++ Libraries	16
3	Communication	19
3.1	Need for Communication	19
3.2	Communication in a team	19

CONTENTS

3.3	Communication across robots	20
3.4	Communication in Robocup	20
4	Narukom	23
4.1	Name	23
4.2	Concept	23
4.3	Architecture	24
4.4	Data Serialization	25
4.5	Data Synchronization	27
4.6	Communication within and across nodes	28
4.7	Communication Channels	28
5	From Theory To Practice	31
5.1	Understanding the basics	31
5.1.1	Messages	31
5.1.2	MessageBuffer	32
5.2	Publish/Subscribe system	32
5.2.1	Message Queue	34
5.3	Blackboard and Synchronization	34
5.4	Catalog Module	35
5.5	Network Communication	37
5.5.1	Network Channel	38
5.5.2	UDP Multicast Channel	38
6	Results	41
6.1	Validation	41
6.2	Synchronization	42
6.3	User Friendliness	44
7	Related Work	45
7.1	Shared Memory	45
7.2	Blackboard/Shared World Model	46
7.3	Message queue and Streams	46
7.4	Publisher/Subscribe	47
7.5	Discussion	47

CONTENTS

8	Future Work	49
8.1	Alternative Channels	49
8.2	Narukom outside RoboCup	49
8.3	Optimization and Debugging	50
9	Conclusion	51
9.1	Out To The Real World	51
	References	54

CONTENTS

List of Figures

2.1	Standard Platform League at RoboCup 2009 in Graz, Austria. . .	6
2.2	3D Simulation League.	7
2.3	Small Size League at Robocup 2009 in Graz, Austria.	8
2.4	Middle Size League game at RoboCup German Open 2009.	9
2.5	Humanoid League at Robocup 2009 in Graz, Austria.	10
2.6	Kouretes at RoboCup 2009. From left to right in the front row are Eleftherios Chatzilaris (SPL), Evangelos Vazaios (SPL), Alexandros Paraschos (SPL), and in the back row Professor Michail G. Lagoudakis (Kouretes Team Leader), Walid Soulakis (Webots Simulation), Professor Nikos Vlassis (Kouretes Team Leader), and Jason Pazis (SPL). . .	11
2.7	Nao's field of view.	12
4.1	Narukom's architecture.	25
5.1	Narukom's publish/subscribe implementation.	33
6.1	Pinger (back), Ponger (middle), Scorekeeper (front) on three different machines.	42
6.2	Data Synchronization	43

LIST OF FIGURES

Chapter 1

Introduction

Communication is a much needed skill, which is met everywhere around us from a human cell and micro-organisms to vastly advanced animals and humans. Even the nature reinvents itself, if a communication channel is broken down. Humans, since the advent of their history, form all kinds of groups striving to achieve a common goal. Especially, in our time and age, they form teams participating in games, where success can only be achieved through collaborative and coordinated efforts. Teams lacking means of communication are doomed to act simply as a collection of individuals with no added benefit, other than the multiplicity of individual skills. For human teams, communication is second nature; it is used not only for teamwork in games, but also in all aspects of life, and takes a multitude of different forms (oral, written, gestural, visual, auditory). Robots can hardly replicate all human communication means, since these would require extremely accurate and robust perceptual and action abilities on each robot. Fortunately, most modern robots are capable of communicating over data networks, an ability which is not available to their human counterparts. However, exploiting such networking means for team communication purposes in an efficient manner that does not drain the underlying resources and provides transparent exchange of information in real time is a rather challenging problem.

1. INTRODUCTION

1.1 Thesis Contribution

This thesis describes a distributed communication framework for robotic teams, which was originally developed for the Standard Platform League (SPL) of the RoboCup (robotic soccer) competition, but can easily serve any other domain with similar communication needs. During a RoboCup game, robot players occasionally need to share perceptual, strategic, and other team-related information with their team-mates in order to coordinate their efforts. In addition, during development and debugging, human researchers need to be in direct contact with their robots for monitoring and modification purposes. Typically, these kinds of communication take place over a wireless network. The naive solution of maintaining a synchronized copy of each robot's data on each node on the network is both inefficient and unrealistic for such real-time systems.

Our proposal suggests a distributed and transparent communication framework, called *Narukom*, whereby any node of the network (robot or remote computer) can access on demand any data available on some other node in a natural and straightforward way. The framework is based on the publish/subscribe paradigm and provides maximal decoupling not only between nodes, but also between threads running on the same node. As a result, *Narukom* offers a uniform communication mechanism between different threads of execution on the same or on different machines. The data shared between the nodes of the team are stored on local blackboards which are transparently accessible from all nodes. In real-time systems, such as robotic teams, data come in streams and are being refreshed regularly, therefore it is important to communicate the latest data or data with a particular time stamp among the robots. To address such synchronization needs, we have integrated temporal information into the meta-data of the underlying messages exchanged over the network. *Narukom*'s distributed nature and platform independence make it an ideal base for the development of complex team strategies that require tight coordination, but also for the distribution of resource-intensive computations, such as learning experiments, over several (robot and non-robot) nodes.

1.2 Thesis Outline

Chapter 2 provides some background information on the RoboCup Competition and the underlying technologies used to develop *Narukom*. In Chapter 3 we state the problem we study and we demonstrate some important aspects of the problem of communication across robots. Continuing to Chapter 4, the core ideas and an outline of the architecture of our proposal are discussed. Moving on to Chapter 5, a thorough discussion about optimization, implementation design and decisions taken during the development of *Narukom* is provided. In Chapter 6 we discuss our results after conducting several experiments in order to evaluate our work. The following Chapter 7 presents similar systems developed by other RoboCup teams, including a brief comparison between those systems and ours. Future work and proposals on extending and improving our framework are the subject of the Chapter 8. The last Chapter 9 serves as an epilogue to this thesis, including a small overview of the system and some long terms plans about *Narukom*.

1. INTRODUCTION

Chapter 2

Background

2.1 RoboCup Competition

The RoboCup Competition, in its short history, has grown to a well-established annual event bringing together the best robotics researchers from all over the world. The initial conception by Hiroaki Kitano (1) in 1993 led to the formation of the RoboCup Federation with a bold vision: “By the year 2050, to develop a team of fully autonomous humanoid robots that can win against the human world soccer champions”. The uniqueness of RoboCup stems from the real-world challenge it poses, whereby the core problems of robotics (perception, cognition, action, coordination) must be addressed simultaneously under real-time constraints. The proposed solutions are tested on a common benchmark environment through soccer games in various leagues, with the goal of promoting the best approaches and ultimately advancing the state-of-the-art in the area.

The RoboCup Soccer event is the flagship competition with the most fans. In this domain, researchers exploit their technical knowledge in order to prepare the best robotic soccer team among all participants. Beyond soccer, RoboCup now includes also competitions in search-and-rescue missions (RoboRescue), home-keeping tasks (RoboCup@Home), robotic performances (RoboDance), and simplified soccer leagues for K-12 students (RoboCup Junior). Broadening the research areas where RoboCup focuses was a very interesting and clever addition, which enables more scientists and researchers to combine their expertise in order to solve real-world problems. A lot of progress has been made so far in many

2. BACKGROUND

disciplines of robotics and RoboCup has been established as one of the most important events around the world.

2.1.1 The Standard Platform League

The Standard Platform League (SPL) of the RoboCup competition is the most popular league, featuring three humanoid Aldebaran Nao robot players in each team (Figure 2.1). This league was formerly known as the Four-Legged League with Sony Aibo robots, which were replaced in 2008 by Aldebaran Nao robots. Games take place in a $4m \times 6m$ field marked with thick white lines on a green carpet. The two colored goals (sky-blue and yellow) also serve as landmarks for localizing the robots in the field. Each game consists of two 10-minute halves and teams switch colors and sides at halftime. There are several rules enforced by human referees during the game. For example, a player is punished with a 30-seconds removal from the field if he performs an illegal action, such as pushing an opponent for more than three seconds, grabbing the ball between his legs for more than three seconds, or entering his own goal area as a defender.



Figure 2.1: Standard Platform League at RoboCup 2009 in Graz, Austria.

The main characteristic of the Standard Platform League is that no hardware changes are allowed; all teams use the exact same robotic platform and differ only in terms of their software. This convention leads to the league's enrichment with a unique set of features: autonomous player operation, vision-based perception,

legged locomotion and action. Given that the underlying robotic hardware is common for all competing teams, research effort has focused on the development of more efficient algorithms and techniques for visual perception, active localization, omni-directional motion, skill learning, and coordination strategies. During the course of the years, one could easily notice a clear progress in all research directions.

2.1.2 Simulation League

Every year there is a number of simulation games taking place in RoboCup competitions. These include 2D soccer games, where teams consist of 11 agents providing the developers with a perfect multi-agent environment to tune and benchmark their solutions. They also include 3D simulation games (Figure 2.2), where the usage of physics engines demands more realistic approaches. Simulators offer the ability to control the amount of “negative” realism added to these environments; thus, it is a great way to allow researchers to focus on multi-agent cooperation approaches and other state-of-the-art algorithms, abstracting from real-world problems (gravity, forces, etc).



Figure 2.2: 3D Simulation League.

2. BACKGROUND

2.1.3 Small Size League

A Small Size robot soccer game (Figure 2.3) takes place between two teams of five robots each. Each robot must fit within an 180mm diameter circle and must be no higher than 15cm, unless they use on-board vision. Robots play soccer on a 6.05m long by 4.05m wide, green carpeted field with an orange golf ball. Vision information is either processed on-board the robot or is transmitted back to the off-field PC. Another off-field PC is being used to communicate referee commands and position information to the robots, when an extra camera mounted on top of the field serves as the vision sensor. Typically, off-field PCs are used in the coordination and control of the robots. Communication is wireless and typically uses dedicated commercial FM transmitter/receiver units.



Figure 2.3: Small Size League at Robocup 2009 in Graz, Austria.

2.1.4 Middle Size League

The Middle Size league is more competitive and demanding, having the largest field dimensions among other leagues (Figure 2.4). Two teams of mid-sized robots consisting of 5 players each with all sensors on-board play soccer on a field of dimensions $18m \times 12m$, whereas relevant objects are distinguished by colors only. Communication among robots (if any) is supported by wireless communications. Once again, no external intervention by humans is allowed, except to insert or remove robots in/from the field. These robots are the best players far among

other leagues. The robots' bodies are heavy enough having powerful motors, heavy batteries, omni-directional camera, and a full laptop computer running in every robot; characteristics that make this league a great domain for research.



Figure 2.4: Middle Size League game at RoboCup German Open 2009.

2.1.5 Humanoid League

The Humanoid League is one of the most dynamically progressing leagues and the one closest to the 2050's goal. In this league, custom-made autonomous robots with a human-like body plan and human-like senses play soccer against each other. In addition to soccer competitions, technical challenges take place. The robots are divided into the size classes: KidSize (30-60cm height) (Figure 2.5), Teen-Size (100-160cm height) and AdultSize (130cm and taller). Dynamic walking, running, and kicking the ball while maintaining balance, visual perception of the ball, other players, and the field, self-localization, and team play are among the many research issues investigated in this league. Several of the best autonomous humanoid robots in the world compete in the RoboCup Humanoid League.

2. BACKGROUND



Figure 2.5: Humanoid League at Robocup 2009 in Graz, Austria.

2.2 Team Kouretes

Team *Kouretes*, is a collaboration of the Intelligent Systems Laboratory at the Department of Electronic and Computer Engineering and the Intelligent Systems and Robotics Laboratory at the Department of Production Engineering and Management. It was the first Greek team ever participating in Robocup competitions, and specializes in the Standard Platform League and the MSRS Simulation League. The team's name, *Kouretes*, comes from the Ancient Greek Mythology. Our four AIBO and four Nao are robots named Epimedes, Paionaios, Iasios, and Idas, after the five Kouretes brothers, who were ancient Cretan warriors. The fifth brother, Hercules, represents all the human members of the team (Figure 2.6).

Kouretes have participated in many competitions and exhibitions, but the highlights of the steep team orbit were the second place in Robocup 2007, Atlanta, USA in the MSRS Simulation League and the first and third place in Robocup 2008, Suzhou, China in the MSRS Simulation League, and the Standard Platform League accordingly.



Figure 2.6: Kouretes at RoboCup 2009. From left to right in the front row are Eleftherios Chatzilaris (SPL), Evangelos Vazaios (SPL), Alexandros Paraschos (SPL), and in the back row Professor Michail G. Lagoudakis (Kouretes Team Leader), Walid Soulaakis (Webots Simulation), Professor Nikos Vlassis (Kouretes Team Leader), and Jason Pazis (SPL).

2.3 The Aldebaran Nao Robot

Aldebaran Robotics designed and assembled a low-cost, easily programmable robot, which can be a great, not only scientific, but also entertainment platform, focusing on the wide audience of robotics' researchers and fans. The development of humanoid robots is a tough initiative that only few universities and companies have undertaken.

Nao, in its current V3+ version, is a 58 cm, 4.3 Kg humanoid robot. To this time, Nao has not been released commercially yet, however Aldebaran's goal is to eventually promote Nao as an educational robotic platform and a family entertainment robot affordable to most budgets. The Nao robot carries a full computer on board with an x86 AMD Geode processor at 500 MHz, 256 MB

2. BACKGROUND

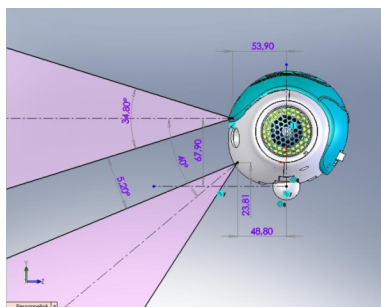


Figure 2.7: Nao's field of view.

SDRAM, and 2 GB flash memory running an Embedded Linux distribution. It is powered by a 6-cell Lithium-Ion battery which provides about 45 minutes of continuous operation and communicates with remote computers via an IEEE 802.11g wireless or a wired ethernet link.

2.3.1 Sensors and Actuators

The Nao V3+ robot features a variety of sensors and actuators. Two cameras are mounted on the head in vertical alignment providing non-overlapping views of the lower and distant frontal areas (Figure 2.7), but only one is active each time and the view can be switched from one to the other almost instantaneously. Each camera is a 30fps, 640×480 color camera, provided with a rich API, allowing many interventions in order to get the best result under any reasonable lighting conditions. A pair of microphones allows for stereo audio perception. Two ultrasound sensors on the chest allow Nao to sense obstacles in front of it and a rich inertial unit (a 2-axis gyroscope and a 3-axis accelerometer) in the torso provides real-time information about its instantaneous body movements. Finally, an array of force sensitive resistors on each foot delivers feedback on the forces applied to the feet, while encoders on all servos record the actual joint position at each time and two bumpers on the feet provide information on collisions of the feet with obstacles.

The Nao robot has a total of 21 degrees of freedom; 4 in each arm, 5 in each leg, 2 in the head, and 1 in the pelvis (there are 2 pelvis joints which are coupled

together on one servo and cannot move independently). Stereo loudspeakers and a series of LEDs complement its motion capabilities with auditory and visual actions.

2.3.2 NaoQi

The Nao programming environment is based on the proprietary NaoQi framework which serves as a middleware between the robot and high-level languages, such as C, C++, and Python. NaoQi offers a distributed programming and debugging environment which can run natively on the robot or remotely on a computer offering an abstraction for event-based, parallel, and sequential execution. Its architecture is based on modules and brokers which can be executed onboard the robot or remotely and allows the seamless integration of various heterogeneous components, including proprietary and custom-made functionality.

2.4 Communication Technology

2.4.1 UDP

The User Datagram Protocol (UDP) is one of the many protocols used in the Internet. Applications can use UDP to exchange messages (called datagrams, in UDP terminology) without creating data-paths or other special transmission channels. UDP is an unreliable service, as it does not offer any guarantee in terms of reliability, ordering, and data integrity. Thus, datagrams can arrive out of order, get duplicated, or get lost without notice. UDP's assumption is that either error checking is not necessary or it is performed by the receiving application. Despite its shortcomings, UDP is used widely in real-time applications as dropping packages is a more viable policy than blocking an application until an expected message is delivered. Another characteristic of UDP is that it has no congestion control, so it does not self-regulate itself. As a result, many applications for on-line gaming or VoIP use UDP to transfer their data. UDP also allows packet broadcasting (packets sent to all nodes in a local network) and multicasting (packets sent to members of a group). Multicast is a one-to-many

2. BACKGROUND

technique for communication over an IP infrastructure network. Senders do not have prior information about how many receivers there are or who they are. Multicast utilizes the network infrastructure by requiring the source to send a packet only once, even if there are multiple recipients, and then the network nodes are responsible for relaying the messages, if necessary, to reach all recipients.

2.4.2 The Publish/Subscribe Paradigm

The publish/subscribe paradigm (2) is a messaging paradigm, where publishers (senders) are ignorant of the recipients of the published messages. Publishers classify each message into a number of classes. On the other end, subscribers express interest on one or more of these classes, so they receive messages only from those classes they are interested in, without any involvement of the senders. There are two widely-used forms for filtering the messages arriving to subscribers: content-based and topic-based. In content-based filtering, a message is delivered to a subscriber, only if certain attributes of the content of the message satisfy the constraints imposed by the subscriber when subscribed. On the other hand, in topic-based filtering, publishers tag their messages with a topic and subscribers subscribe to certain topics which they know they are interested in. There is also a hybrid approach, whereby publishers tag their messages with topics and subscribers create content-based subscriptions. Many publish/subscribe systems use an intermediate broker for dispatching the messages. The broker is responsible for maintaining the list of subscriptions and forwarding all the published messages to the appropriate recipients.

2.4.3 Google Protocol Buffers

Google Protocol Buffers (3) are a flexible, efficient, automated mechanism for serializing structured data. The user defines the data structure once and then uses the generated source code to write and read the defined structures to and from a variety of data streams using a variety of programming languages. Another great advantage of protocol buffers is that data structures can be updated without breaking the already deployed programs, which are capable of handling the old format of the structures. To use protocol buffers one must describe the

information for serialization by defining protocol buffer messages in `.proto` files. A protocol buffer message is a small record of information, containing name-value pairs. The protocol buffer message format is simple and flexible. Each message type has at least one numbered field. Each field has a name and a value type. The supported types are integer, floating-point, boolean, string, raw bytes, or other complex protocol buffer message types, thus hierarchical structure of data is possible. Additionally, the user can specify rules, if a field is mandatory, optional, or repeated. These rules enforce both the existence and multiplicity of each field inside the message. As a next step, the user generates code for the desired language by running the protocol buffer compiler. The compiler produces data access classes and provides accessors and mutators for each field, as well as serialization/unserialization methods to/from raw bytes. Officially, Google supports C++, Java, and Python for code generation, but there are several other unofficially supported languages.

2.4.4 Blackboard

Blackboard (4) is a software architecture model, in which multiple individual knowledge sources share a common knowledge base, called *blackboard*. The sources can read or update the contents of the blackboard and therefore, as a result, the sources could cooperate to solve a problem. The Blackboard model was introduced in order to handle complex, ill-defined problems, whose solution is the sum of the solutions of simpler problems. Typically, a blackboard system consists of three main components:

The Knowledge Sources These are modules that perform certain tasks and contribute to the solution of the problem. Knowledge sources are software programs which operate independently of each other, which enables the designer of an application to eventually replace certain sources that become outdated or slow down the system. This is an important characteristic especially nowadays when the pace of developing software is rapid and the necessity of updating individual modules is invaluable.

2. BACKGROUND

The Blackboard This is a shared memory area where partial solutions, data, and any other information contributed by the knowledge sources is stored and becomes available for access by all knowledge sources. Importantly, the blackboard can be used as a communication mechanism between the sources.

The Control Component This is a module that acts as a moderator between the knowledge sources in order to organize them as effectively as possible.

2.4.5 Monitor

In concurrent programming, a monitor (5) is a thread-safe object. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This characteristic simplifies the implementation of monitors compared to code that may be executed in parallel. Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met. Monitors were invented by C.A.R. Hoare and Per Brinch Hansen, and were first implemented in Brinch Hansen's Concurrent Pascal language.

2.4.6 Boost C++ Libraries

The Boost libraries (6) are free, peer-reviewed, portable C++ source libraries. Boost libraries are intended to be widely useful and usable across a broad spectrum of applications. Boost Libraries range from general-purpose libraries, like the `smart_ptr` library to operating system abstractions, like the `Boost FileSystem`, to libraries primarily aimed at other library developers and advanced C++ users, like the meta-programming template (MPL) and the Domain Specific Language creation (Proto). In order to ensure efficiency and flexibility, Boost makes extensive use of templates. Boost has been a source of extensive work and research into generic programming and meta-programming in C++.

The current Boost release contains over 80 individual libraries, including libraries for linear algebra, pseudo-random number generation, multi-threading, regular expressions, unit testing, network programming, and many others. The majority of Boost libraries are header-based, consisting of inline functions and templates, and as such do not need to be build in advance of their usage. It must be noted that Boost Libraries are cross-platform.

2. BACKGROUND

Chapter 3

Communication

3.1 Need for Communication

Communication is the process of transferring information from one entity to another. The need for such a process is obvious as different entities should co-exist and collaborate with each other. Thus, being able to express feelings, desires, needs and information is crucial to all living beings from a tiny cell to a human being. People instinctively gather and form groups with a common goal, either important (i.e. survival) or not (i.e. soccer). The integrity of these groups is proportional to the communication between its members.

3.2 Communication in a team

Communication among team members is a crucial ability for any kind of team striving to achieve a common goal, especially for teams participating in games, where success can only be achieved through collaborative and coordinated efforts. Teams lacking means of communication are doomed to act simply as a collection of individuals with no added benefit, other than the multiplicity of individual skills. For human teams, communication is second nature; it is used not only for teamwork in games, but also in all aspects of life, and takes a multitude of different forms (oral, written, gestural, visual, auditory).

3. COMMUNICATION

3.3 Communication across robots

Robots can hardly replicate all human communication means, since these would require extremely accurate and robust perceptual and action abilities on each robot. Fortunately, most modern robots are capable of communicating over data networks, an ability which is not available to their human counterparts. However, exploiting such networking means for team communication purposes in an efficient manner, that does not drain the underlying resources and provides transparent exchange of information in real time, is a rather challenging problem. Even if data networks are sufficient for exchanging data between robots and computers, there is a lot of research on other fields such as computer vision or sound processing, which could be useful for communication. Research in these areas aims at creating more human-like robotic platforms, which in the future could be used as a substitute or assistance to humans. This means that, apart from the network packets, robotic systems have to be able to handle other types of messages, like sounds or images. Unification of all these different messages is easier said than done, however a lot of work has been made in order to have a transparent way of dealing with this diversity.

3.4 Communication in Robocup

A key aspect of most RoboCup leagues is the multi-agent environment; each team features 11 (simulation), 5 (small-size), 6 (mid-size), 3 (humanoid), or 3 (standard platform) robots. These robots cannot simply act as individuals; they must focus on teamwork in order to cope effectively with an unknown opponent team and such teamwork requires communication. Fortunately, the use of a wireless network is allowed in most leagues, however there are strict rules about the available bandwidth for each team with the maximum bit rate varying between 500Kbps and 2.2Mbps.

A common requirement in almost all RoboCup leagues is that all players must be able to listen to the so-called game controller (7), which is software that acts as a game referee and runs on a remote computer. The game controller broadcasts messages over the wireless network that contain important information about

the state of the game (initial positioning period, game start, game time, game completion, goal scored, penalty on a player, etc.). Even though this is one-way communication (the game controller does not expect any messages from the robots), the ability to receive and decode correctly game controller messages is a crucial communication component for each team. It has even been suggested that teams not listening to and obeying the game controller of the league should be automatically disqualified.

Each robot of an SPL team typically maintains a local perception obtained through a vision module, a local estimate of self location obtained through a localization module, a local active role which might be dynamically assigned by some behavior module, and a local status which indicates the state of the robot (active, penalized, fallen-down, etc.). To assess the benefits of the ability of transparent intra-team communication, consider the following scenarios: (a) say that robot 3 has spotted the ball, whereas robot 2 is scanning for it; robot 2 could be facilitated in locating the ball through a `perception.robot[3].ball` query to get the (possibly noisy) ball coordinates from robot 3 and actively look for it in that direction; (b) say that robot 2 has lost its estimate of self-location; a `perception.robot[*].teammates` that returns coordinates of visible teammates could greatly facilitate recovery of its own location; (c) say that a robot needs to determine its current role depending on the current formation of the team; a `localization.robot[*].location` query could return the (estimated) locations of all robots in the team and therefore lead to a decision about role assignment; (d) say that the goalie (robot 1) has fallen down leaving the goal unprotected; a `behavior.robot[1].status` could guide a defender to urgently cover the goal in some way. Additionally, the benefits of communication can be valuable to the human developers during debugging: (a) a `perception.robot[3].camera` query could bring the camera stream of robot 3 to a remote computer; (b) a `localization.robot[*].location` could bring the (estimated) locations of all robots for display on a remote monitoring computer; (c) a `behavior.robot[*].status` query could be used to visualize the status of each robot. These examples indicate that development and actual game play can be greatly facilitated by a transparent and efficient communication framework.

3. COMMUNICATION

Chapter 4

Narukom

4.1 Name

Before we dive into the technical details about Narukom, we spare a little time to explain why we gave the name Narukom to our framework. In the beginning, the name was Kouretes System of Communication (KouSoCom), but after watching approximately 300 episodes of the popular Japanese cartoon (a.k.a anime) Naruto in less than ten days, we decided to include the word "Naruto" in the title. As a result KouSoCom became Narukom from the combination of the words Naruto, Kouretes, and Communication.

4.2 Concept

After the adoption of the Nao platform, our team did not have a framework to share information between the robots. The majority of the messages was based on quick hacks developed by different people which made the code maintenance infeasible, thus the need for a well-defined, easy-to-use framework emerged. The proposed communication framework, Narukom, tries to address the communication needs of robotic teams by providing an efficient, flexible, and simple way of exchanging messages between robots, without imposing restrictions on the type of the data transferred over the network.

4.3 Architecture

Narukom's main idea is a topic-based publish/subscribe system for exchanging messages locally or over the network¹. On each node there is only one instance of an intermediate broker (Message Queue), that temporarily stores published messages and delivers copies to the interested subscribers. The Message Queue holds a hierarchical topic tree, which is used for subscribing to messages on a particular topic (`on_topic`) or on a topic and its ancestors (`above_topic`) or on a topic and its descendants (`below_topic`).

A robotic agent is composed of modules that implement different concrete functionalities. The idea that modules produce data for other modules and use data from other modules comes quite natural, as, for example the behavior module depends on data from the localization module and the object recognition module uses color-segmented image from the image segmentation module. In our team's architecture, modules that operate on the same data are grouped in a thread. Threads are perceived as publishers or subscribers (or both), which could publish data on certain topics or receive data by subscribing to certain topics. In order to simplify the way threads or modules share such data, Narukom introduces a distributed Blackboard design, whereby modules from different threads on the same or even different network nodes access transparently the desired data.

A Blackboard module in the context of the publish/subscribe paradigm is simply a subscriber, which serves other modules' requests for data stored internally and published by modules or threads on the same or on a remote node. There could be either one blackboard for all the modules/threads of a robot or multiple blackboards, one for each thread running on the robot. Our team adopted the latter design, because of the synchronization problems occurring on the former. A graphical overview of the Narukom architecture is shown in Figure 4.1. A simple interface is given to the user to access data; requests have the following form: `publisher.robot[X].type_of_message`. This differs from the anticipated form

¹ Apart from robot-to-robot communication, Narukom is used also for robot-to-computer communication, so we refer to both systems (robots and computers) as network nodes, or simply nodes.

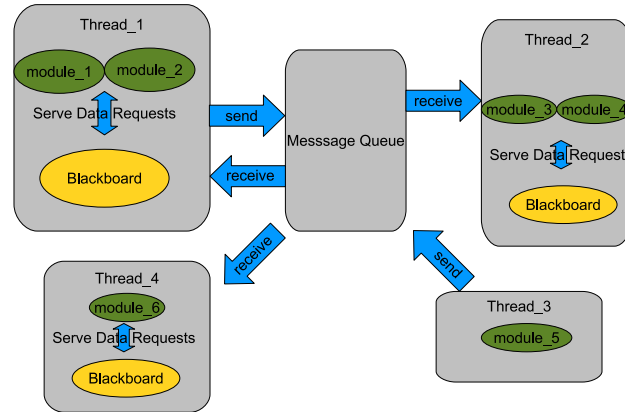


Figure 4.1: Narukom's architecture.

of `robot[X].publisher.type_of_message`, but it was a design decision that will be explained in the next chapter.

4.4 Data Serialization

Serialization is a well-known issue when exchanging data over the network between heterogeneous systems. This issue was a major concern while designing Narukom, consequently several frameworks were examined according to four criteria:

Language support Support for both C++ and Python was a hard constraint as these two languages are now used on the Nao platform.

User friendliness Users should have the ability to describe easily and quickly new data structures.

Cross-platform *Narukom* aims to be a cross-platform framework.

Efficiency The selected framework must be efficient and robust.

There were five main candidate solutions that were extensively reviewed: XML, CORBA, boost serialization library, Google's Protocol Buffers and a custom serialization method. Here, follows a brief comparison between those alternatives

4. NARUKOM

in order to make clear why we chose Protocol Buffers over the other frameworks. XML met all our needs, but efficiency, as extensive parsing of XML files is a resource draining procedure on a real-time system, such as a robotic system. Although CORBA does meet all the criteria, it is not the ideal solution in terms of efficiency. Continuing to the boost library, it failed in the terms of user friendliness requiring users to write their own serialization methods and at the same time there was no simple way to define new messages. The procedure is not tedious, but it was preferred to adopt a solution, where messages could be declared and described independently of a specific programming language. The development of our own serialization method, as tempting as it sounds, it would introduce serious debugging and code maintenance problems, not to mention that optimizing the whole procedure is a long and painful path to follow. Finally, as far as Protocol Buffers are concerned, they cover all our needs. Definition of messages is simple and intuitive, C++ and Python are officially supported by Google, and last, but not least, the code is heavily tested and optimized. All these made Protocol Buffers the ideal tool for constructing our hierarchical message structures. Every message definition in Narukom takes the following form with five mandatory attributes in its preamble:

```
message xxxMessage{
    required string host      = 1 [default = "localhost"];
    required string publisher = 2 [default = "" ];
    required string topic     = 3 [default = "global"];
    required int32  timeout   = 4 [default = 0];
    required string timestamp = 5 [default = ""];
    < attributes specific to the xxxMessage type >
}
```

These five attributes can be characterized as the metadata of every message required by Narukom in order to provide its services accurately. A short explanation of each of these attributes follows:

- **host** is the unique name of the node sending the message
- **publisher** is the name of the publisher that generated the message

- `topic` is the topic to which the message belongs
- `timeout` is a 32-bit integer indicating the amount of time (in milliseconds) after which the message expires and should be discarded from the Blackboard
- `timestamp` is a string indicating the absolute time at which the message was published and is used for synchronization purposes between publishers and subscribers.

4.5 Data Synchronization

Synchronization among the modules of a robot is vital for accurate operation². Narukom, in order to address the decoupling between publishers and subscribers, adds some synchronization information in the meta-data of each message (`timeout` and `timestamp`). Under this convention, the Blackboard's original interface had to be slightly altered to provide modules with synchronized data, should the need arises. If a module queries the Blackboard with a certain time stamp value, the Blackboard will have to find and return the data with the closest `timestamp` value in the Blackboard. The `timestamp` attribute is the absolute time at which the message was published to the Message Queue and the `timeout` attribute is the (relative) amount of time (in milliseconds) indicating how long a subscriber should keep the message in its own Blackboard, counting from the time of arrival. The timeout mechanism is necessary for avoiding excess amounts of expired, useless data on the Blackboard. It should be stretched that these synchronization rules cannot be imposed and every subscriber has the flexibility to comply or not with those rules. Narukom's distributed Blackboard implementation utilizes these two attributes in each message to achieve a balance between the length of the synchronization period and the amount of memory needed for this purpose, allowing different settings for each message type.

²Anecdotal evidence for this claim includes our team's performance at RoboCup 2009, where the camera images processed by our vision module were not in synch with the robot head pose recorded by the behavior module, thus yielding erroneous object perceptions with catastrophic consequences on the robot's behavior.

4.6 Communication within and across nodes

Communication between threads running on the same node is implemented simply through the Message Queue of the node, as described already above. Communication between different nodes adopts a somewhat different approach. Given the publish/subscribe nature of Narukom, it is clear that the only missing part for communication over the network is a module, which subscribes to all topics and broadcasts over the network messages destined for other nodes, and at the same time listens to broadcast traffic on the network and publishes the relevant messages on the local Message Queue. To this end, Narukom introduces the Catalog module. This module maintains contact information about the other nodes on the network and the type of data other nodes might be interested in. The question we should address is why it is important to have this functionality. In a distributed system, especially in a robotic environment, agents must be aware of all the on-line team members in order to utilize all the available resources for exchanging data across nodes and facilitating the development of complex behaviors and distributed algorithms. Moreover, Catalog Module provides Narukom with extendibility, as user-defined channels can be used as part of Narukom's infrastructure.

4.7 Communication Channels

A multi-purpose robot, such as Nao, has multiple means of interacting with its environment, either sensors or actuators. Sensors (camera, network adapter, microphones, etc.) sense the robot's environment, whereas actuators (motors, network adapter, speakers, etc.) affect the robot's environment. If the effect of a robot's actuator is observable by another robot's sensor, this pair of actuator/sensor offers a possible communication channel. Narukom tries to integrate different channels into the framework. To accomplish that, it introduces a generalized Catalog module, which is responsible for holding contact information about any available peer in Narukom's framework and at the same time is responsible for controlling all the channels available between peers, which are also implemented in Narukom.

4.7 Communication Channels

By default, the Catalog module establishes two channels: one listening for network-broadcast data from other peers and another one for broadcasting data to all the reachable peers through the network. Both channels use the UDP transmission protocol. The Catalog module establishes an extra incoming channel for interacting specifically with the game controller, wherever there is such a need.

4. NARUKOM

Chapter 5

From Theory To Practice

This section describes all the optimizations, implementation decisions, and details of Narukom. The primary goal of Narukom is to be used as a communication framework in robotic environments, so most of our choices were in favour of real-time embedded systems focusing on the needs of the Standard Platform League environment. However, those implementation choices do not restrict Narukom's possible uses.

5.1 Understanding the basics

5.1.1 Messages

The exchanged data between the threads are protocol buffers messages defined by the user. Narukom requires to include in each message definition some attributes that are of outmost importance in order for Narukom to work properly and handle every message. As mentioned before, every message definition in Narukom takes the following form with five mandatory attributes in its preamble:

```
message xxxMessage{
    required string host      = 1 [default = "localhost"];
    required string publisher = 2 [default = "" ];
    required string topic     = 3 [default = "global"];
    required int32  timeout   = 4 [default = 0];
    required string timestamp = 5 [default = ""];
```

5. FROM THEORY TO PRACTICE

```
< attributes specific to the xxxMessage type >  
}
```

Narukom includes a definition of a basic structure called `BasicMessage` in order to deal with all the messages. Each attribute serves a purpose,

- `host`, `publisher` are attributes used to distinguish messages from each other,
- `topic` is the topic in which each message should be published, and
- `timeout`, `timestamp` is the temporal information needed for synchronization purposes.

5.1.2 MessageBuffer

The Message Buffer is thread-safe structure and, as its name suggests, is a buffer in which protocol buffer messages are stored. Apart from the simple vector that holds the messages, the structure has a string attribute named `owner`, which denotes the name of the owner of the buffer. It must be noted that a message buffer is implemented as a monitor.

5.2 Publish/Subscribe system

Our Publish/Subscribe system implementation is shown in Figure 5.1. Both publishers and subscribers have one message buffer through which the exchange of data takes place. The Message queue periodically checks all the publishers' buffers and copies them to the buffers of all interested subscribers. This approach minimizes the race conditions when accessing messages, at the cost of high decoupling between the publishers and subscribers, which in real-time systems could be a source of undesirable behavior. In the next section, it is described how this problem is addressed. Continuing with the publish/subscribe system, classes that use the infrastructure provided by Narukom, should inherit from either the `Publisher` or the `Subscriber` class (or from both). Although there is a default implementation of the interfaces for both classes, it is recommended for derived

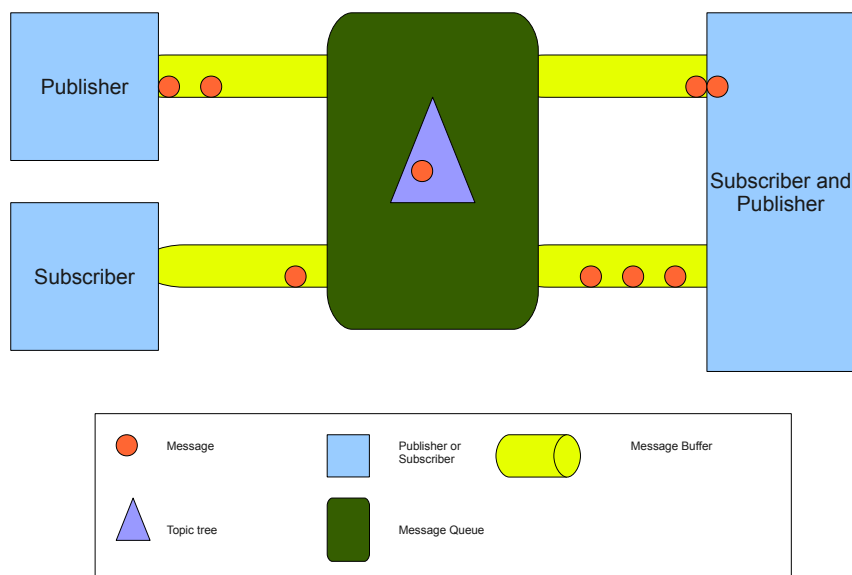


Figure 5.1: Narukom's publish/subscribe implementation.

classes not to rely on the default implementation as it might have undesired effects on the behavior of the system. The `Publisher` class requires the implementation of the method:

```
void publish(google::protobuf::Message* msg)
```

The default implementation of this method just adds the message `msg` to the buffer, which in the majority of the scenarios would be sufficient for sending a message. On the other hand, classes derived from the `Subscriber` class are strongly encouraged to provide their own implementation of the method:

```
void process_messages()
```

The default implementation of `process_messages()` is provided just as a reference. Through this function incoming messages should be read and processed.

5. FROM THEORY TO PRACTICE

5.2.1 Message Queue

Message queue performance affects directly the overall performance of our system, thus several optimizations have been made to this class. The most important optimization is that the internal topic tree is used as a reference structure, while the actual data structure used is a hash table on all the topics included in the tree. The configuration of the tree is done via an XML file on each node; at the initialization of the message queue the file is parsed and the tree is created. The hash table allows for quick retrieval of the list of subscribers to which the arrived message must be delivered. Furthermore, when it comes to reading the message's meta-data, a cast to `BasicMessage` type is used, overriding the normal way of accessing data in protocol buffers. It must be stressed that this approach is used **ONLY** for reading and **ONLY** in **Blackboard and Message Queue** classes. **Under no circumstances is it recommended as the normal way for accessing the meta-data required by Narukom.** This is considered a hardcore optimization which could stop working after a new release of Protocol Buffers. Last, but not least, instead of keeping pointers to the publisher or subscriber, a message buffer pointer is used in order to avoid the small overhead of calling an extra method every time the `Message Queue` needs to access a buffer. As insignificant as these optimizations may seem, in SPL, where software makes the difference, they could decide the winner of a game.

5.3 Blackboard and Synchronization

The implemented publish/subscribe system maximizes the decoupling between publishers and subscribers, thus many synchronization issues may arise between threads running locally on a node. This problem is dealt with by adding two attributes in the meta-data of each message `timeout` and `timestamp`. Narukom introduces some conventions for each message: (a) when a message is published a timestamp is assigned (at least in the default implementation of `publish`), (b) when a subscriber receives a message she should check the validity of the data by ensuring that those data are not expired, and (c) on data expiration subscribers are advised to dispose those messages in order to limit the excess and

useless information on a node. It should be noted that these are conventions of Narukom that are used internally and cannot be imposed on subscribers; each subscriber is free to comply or not with them. Narukom's blackboard utilizes the time information to minimize memory allocation and provide synchronization services to the higher-level system. The Blackboard implementation creates two indexes over the stored messages, one to provide fast access for data requests and another to provide fast cleanup time. The latter index is an ordered list in terms of absolute timeout and is calculated whenever new messages are stored in the blackboard. Frequently, the blackboard cleans up all the expired data.

5.4 Catalog Module

Communication between the nodes of Narukom is achieved through channels, which are well-defined pairings between actuators (motors, speakers, network adapter) and sensors (camera, microphones, network adapters). Narukom was designed to be able to utilize all the individual channels and integrate those channels in the framework.

The Catalog module could be characterized as the address book of each node in Narukom, since it holds all the contact information of all the reachable peers by the node itself. Furthermore, the Catalog module is responsible for creating and destroying the various channels that the node is able to use in order to communicate with other nodes. The idea behind this module is that in distributed environments a node must be aware of all the available nodes and what type of services they provide. In detail, the information stored in a Catalog module includes the names of all reachable peers, the channels through which these nodes are available, and the kind of data (topics) they are interested in. Moreover, the Catalog module holds a list of all the usable channels on the node and has the ability to start, stop, or filter the type of messages that are sent through each channel. The filtering is done by sending commands that a channel can interpret as one of the above functions. These commands consist of a Protocol Buffers message with the following attributes

```
message ChannelCommand{
```

5. FROM THEORY TO PRACTICE

```
required string type      = 1 [default = ""];
required string element = 2 [ default = "none"];
required string options = 3 [ default = "none"];
}
```

There are four different types of commands

- **subscribe**: when this type of message is received, a channel must subscribe to a certain topic.
- **unsubscribe**: when this type of command is received by a channel, it must unsubscribe from a certain topic.
- **enable**: this command is used to enable a specific type of message to be sent through the channel; by default after the subscription of a channel all the incoming messages are enabled.
- **disable**: obviously, this is the opposite of the enable command.

The attribute `element` is in accordance with the command type. In case of (un)subscribe, it should hold the topic, whereas when sending an **enable** or **disable** command it must be the type of message to be enabled or disabled. Channels should implement a simple interface, which is:

- `bool start()`: with this function the Catalog module starts a channel.
- `bool stop()`: this function is used to stop a channel.
- `list_of_hosts available_peers()`: the Catalog module asks for all the information about the reachable peers with this function.
- `list_of_strings preferred_message()`: with this function the Catalog module is informed about what messages could be produced from this channel.
- `bool process_command(const ChannelCommand cmd)`: this function is used to send a command over a channel.

These four functions are adequate for Catalog to control a channel and provide the system with all the required information about a channel. Before closing this section, it should be noted that all these are conventions are used by Narukom to provide users with a unified way of controlling the behavior of a node in terms of communication with other reachable nodes. Narukom does not require this functionality, but should a programmer decide to extend Narukom's capabilities by creating a customized channel, this channel should be treated as an integrated part of the rest of the system. However, there could be a channel which does not implement the required interface; this does not raise any problem for the rest of the system, as it would be treated as any other module on the system. Finally, the options attribute is used for additional information that may be required for example, in case of **subscribe** what mode should be used (**on_topic**, **below_topic**, **above_topic**).

5.5 Network Communication

Network communication is the first and most important way of exchanging data between nodes. Protocol buffers messages are serialized and if their size makes it impossible to send them as one message they are chopped to several network packets. The definition of the NetworkPacket structure follows:

```
message NetworkHeader
{
  required uint32 message_num = 1 ;
  required uint32 packet_num = 2 [default = 1];
  required uint32 number_of_packets = 3 [default = 1];
  required string timestamp = 4 [default = ""];
  required uint32 timeout = 5 [default = 1000];
  required string host = 6;
  required string type = 7;
}

message NetworkPacket{
  required NetworkHeader header = 1;
```

5. FROM THEORY TO PRACTICE

```
required bytes byte_buffer = 2;
}
```

A network message acts as a buffer for `NetworkPackets`. On the other end, network messages are stored in network message buffers.

5.5.1 Network Channel

The `NetworkChannel` class is a class which encapsulates all the communication functionality provided by any network channel (i.e. TCP-based, UDP-based, etc.). This class is a derived class of `Channel` and provides implementation for all the methods of its base class. In addition, it introduces new functionality provided mainly in a network, such as resolve host or notify on events, like sent or timeout, etc.

5.5.2 UDP Multicast Channel

The only channel that is so far implemented in Narukom is a simple multicast channel over UDP. A sharp reader could easily infer that `UdpMulticastChannel` class is simply a child class of `NetworkChannel`. The implemented channel is not reliable, meaning that there is no guarantee that, when sending a message through this channel it would reach its destination, but provides a partial guarantee that if a part of the message arrives on one node, then the whole message should be delivered. The algorithm that ensures that a whole message will be delivered is the following:

```
receive packet from the network =>
add packet to the appropriate network message =>
Is the network message complete?
a) yes: then recreate the original message
        and publish it then remove it from the buffer
b) no: if the packet is out of order send NACK Messages
        for all the pending packets
```

At first sight, it might be a wrong decision to use an unreliable channel for our transmissions, but in real-time systems the described behavior is preferred

over a bandwidth-consuming approach (i.e. TCP). The `UdpMulticastChannel` subscribes by default to the global topic, where important information interesting to all the nodes on the network is published. Finally, the `Catalog` module under certain circumstances (during a RoboCup game) initializes another UDP Multicast Channel in order to listen to the game controller.

5. FROM THEORY TO PRACTICE

Chapter 6

Results

The evaluation of the proposed approach was made on the following subjects:

Validation To validate that Narukom works properly we carried out experiments testing both inter-process and intra-process communication.

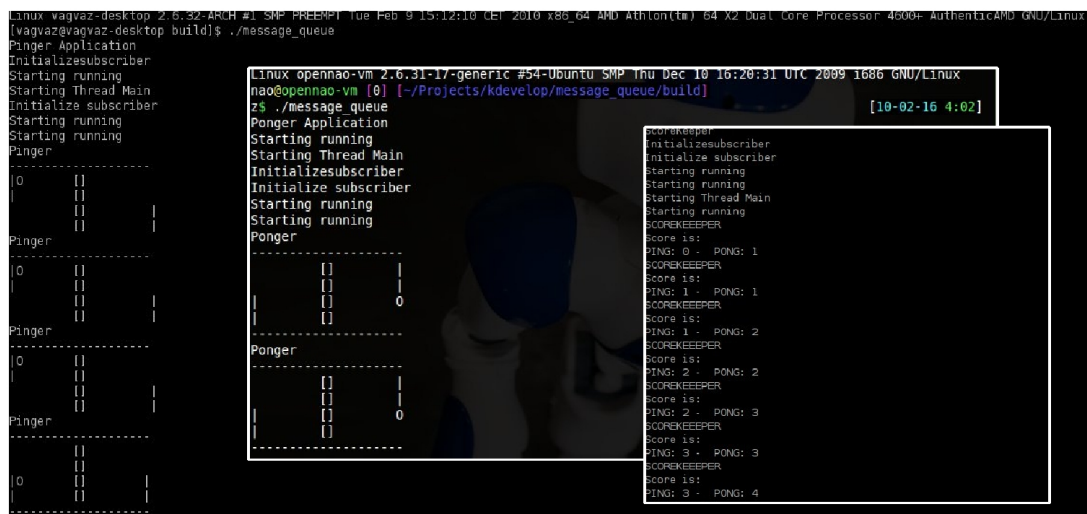
Synchronization In order to test the synchronization capabilities of Narukom a simple multi-threaded application was developed.

User friendliness A small group of developers was asked to create a simple chat application to assess the user-friendliness of our framework.

6.1 Validation

Narukom has been tested successfully for both inter-process communication on a single node and inter-thread communication across different nodes using a simple threaded application which simulates a ping-pong (table tennis) game. There are two threads for the two players and two message types (**Ping** and **Pong**), one for each player. These two threads communicate through Narukom's publish/subscribe infrastructure, which is used to deliver the **Ping** and **Pong** messages between them. Apart from the two players, there is a scorekeeper, another thread which is responsible for keeping the score, as its name suggests. The player who has the ball picks randomly a side (left or right) to which the ball should be sent and publishes a message to declare the completion of his move. On the other side

6. RESULTS



The image displays three terminal windows running on different Linux machines, illustrating the execution of three processes: Ping, Pong, and Scorekeeper. The background of the terminals features a RoboCup logo.

- Left Terminal (Ping):** Shows the execution of the Ping process on a 64-bit native laptop. It includes initialization steps like 'Initialize subscriber' and 'Starting Thread Main', followed by a series of 'Ping' messages and a 'Pong' response.
- Middle Terminal (Pong):** Shows the execution of the Pong process on a 32-bit virtual machine. It includes initialization steps like 'Initialize subscriber' and 'Starting Thread Main', followed by a series of 'Pong' messages and a 'Ping' response.
- Right Terminal (Scorekeeper):** Shows the execution of the Scorekeeper process on a 64-bit native PC. It includes initialization steps like 'Initialize subscriber' and 'Starting Thread Main', followed by a series of 'Score' messages and 'Ping/Pong' responses.

Figure 6.1: Ping (back), Pong (middle), Scorekeeper (front) on three different machines.

of the table, the other player picks the side where his racket will be placed (left or right) and listens for the opponent's move (he subscribes to opponent messages). If the side he chose for the racket matches the side to which the opponent sent the ball, the game continues with the next move. If the sides of the racket and the ball do not match, a point is awarded to the player who sent the ball and the game continues with the next move. The scorekeeper subscribes to both types of messages and continuously prints out the current score. Note that these threads could be running on different nodes and these nodes do not have to be identical platforms. Screenshots of the three processes (ping, pong, scorekeeper) running on three different Linux machines (64-bit native on laptop, 32-bit virtual on laptop, and 64-bit native on pc respectively) communicating using Narukom over a wireless network are shown in Figure 6.1. Live action of this experiment is available at <http://tinyurl.com/robocup2010submission101>.

6.2 Synchronization

In order to ensure that Narukom's synchronization capabilities are adequate for SPL, we developed an application where two threads run at different frequencies

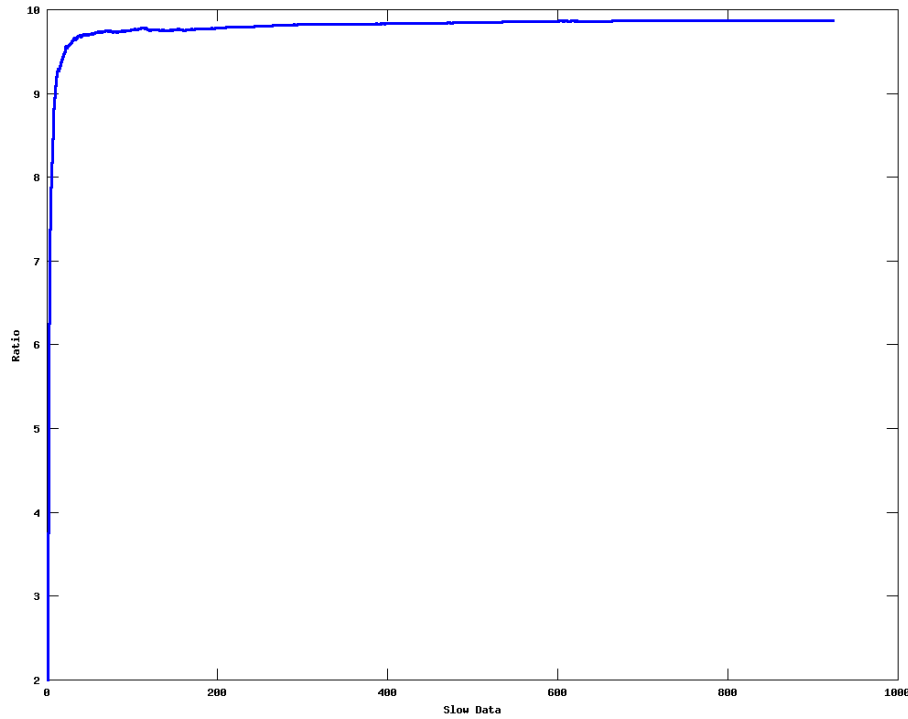


Figure 6.2: Data Synchronization

and a third thread which operates on the data published by those two threads. In more detail, there is one thread (**Fast**) running approximately every 10 milliseconds and another (**Slow**) running every 100 milliseconds. Both threads increase a counter every time they are scheduled. The third thread (**Ratio**) reads the latest available data of *Slow* from the blackboard, then according to the message's timestamp, it queries the blackboard for the produced data by **fast**. Finally, the *Snapshot* outputs the ratio of the two counters. The Figure 6.2 demonstrates the results of the experiment. The Ratio is close to the ideal value 10, the difference can be explained by the fact that the three threads are approximately scheduled on time.

6. RESULTS

6.3 User Friendliness

Nowadays, ease of use plays an important role in the adoption of a product, thus we decided to conduct a limited (in terms of the number of people) survey to assess the level of user friendliness of our framework. After a short presentation of Narukom, we asked a small group of developers to create a simple chat application. Afterwards, we asked them to write a brief description of their thoughts working with Narukom.

The results of this survey were more than satisfactory. All the participants were able to complete the required task within an acceptable time period (1 hour on average). It must be noted that 65% of the participants had little experience with C++, let alone network programming in this language. However, everyone agreed that the framework was easy to use, but it lacked an adequate tutorial. Moreover, after concluding the exercise all participants acquired good understanding of how Narukom works underneath and the services it can (or could in the future) provide, and by suggesting new exciting uses.

Chapter 7

Related Work

In this section we give a short overview of what other RoboCup teams have published to our knowledge about their communication system and compare their work to ours. There are various approaches for sharing information among processes on a robot and among processes on different robots. These approaches can be summarized into the a few main ideas.

7.1 Shared Memory

The main idea here is to define a shared memory area on each node, which holds information about the current world model. Each module can write to and/or read data from this shared location. Under this scheme, different nodes use a simple UDP protocol to exchange important data. This is quite similar to our approach, however communication is mainly point-to-point and lacks the organization and efficiency imposed by the hierarchical topic-based publish/subscribe scheme. Furthermore, depending on the implementation, problems with corrupted data could be appear, if no lock and safety mechanisms are implemented. This idea with variations has been adopted by teams as Austin Villa (8) and BreDoBrothers (9).

7. RELATED WORK

7.2 Blackboard/Shared World Model

This is a similar idea with the previous, however different components access the central storage data via well-defined interfaces, providing the mechanisms that might be absent from the approach above. Team robots share the information of their blackboards creating a joint world model composed of the individual robots' perceptions. There is one variation from GTCMU (10) where validity flags are added to the objects stored in order to declare them as "use with caution" when the objects are not updated for a long period of time. For intra-robot or debugging purposes, custom communication systems are used over UDP or TCP. The purpose of data exchange suggests the protocol used, for example for debugging purposes TCP is used, while when in game UDP transmission is preferred. The shared model is close to Narukom's implementation of distributed blackboard, but the exchange of newly created data requires from the programmer writing code for sending and receiving those data, whereas on Narukom newly declared Messages are immediately ready for network transfer. The vast majority of RoboCup teams, such as Zadeat (11) and rUNSWift (12) use a slight variation of this approach.

7.3 Message queue and Streams

This idea has been advocated recently by team B-Human (13) and earlier by the German Team (14). It is based on using message queues for constructing three different types of cross-platform communication streams: (a) inter-process communication between the cognition and the motion module, (b) debug communication between robots and remote computers for debugging purposes, and (c) information exchange between the robots. The first type is similar to Narukom's inter-process communication. The other two types are implemented differently in Narukom, but they serve the same purposes. Again, this idea lacks the organization and efficiency of Narukom. It must be noted, however that a lot of effort has been put into creating a fully-featured Stream library to support the required functionality, whereas in Narukom this functionality is provided for free through the Google protocol buffers.

7.4 Publisher/Subscribe

This idea is based on the publish/subscribe paradigm and has been adopted only the Austrian Kangaroos (15) team. The idea is to use a publish/subscribe architecture for exchanging data between the threads on the same machine by triggering signals in all connected modules, if a subscribed memory location is altered. This design allows an IRQ-like implementation of service calls at a high system level. Network communication is done via SOAP wrapped functions calls, which generally are considered heavy compared to Narukom's approach. On the other hand, in this approach each module has a more direct access to remote data, simply by subscribing to remote memory locations.

7.5 Discussion

Generally speaking, Narukom integrates features that support efficiently any communication need in the SPL; these features are not found collectively in any other existing communication system. One may argue that Narukom's functionality could be implemented using the infrastructure provided by the proprietary NaoQi middle-ware offered by Aldebaran Robotics with the Nao robots. This is true, however Narukom's purpose is to provide a communication system that does not depend on proprietary technology and can be used widely on many different robotic teams, apart from Nao robots, and other real-time distributed systems. Narukom also represents our efforts in departing from the dependence on the NaoQi framework towards platform-independent customizable software architecture for robotic teams.

7. RELATED WORK

Chapter 8

Future Work

Our work is just a small, but well-founded, step towards our notion of the ideal communication framework for robotic teams and other similar distributed systems. Thus, there are plans for extending, adding, and optimizing further the proposed framework.

8.1 Alternative Channels

The typical communication channel is certainly the one that utilizes the network adapter as both sensor and actuator and this is the only communication channel currently implemented in Narukom. However, other communication channels can be realized using other pairings, for example messages encoded into motions using the sender's joints and decoded through the receiver's camera or messages encoded into sounds using the sender's speakers and decoded through the receiver's microphones. Narukom can treat all these channels transparently as long as appropriate coders/decoders (drivers) for these channels are provided. Despite the great difficulty associated with the implementation of such interfaces, their importance to true human-like robot autonomy is clearly high.

8.2 Narukom outside RoboCup

The use of Narukom outside the context of Robocup is also an important goal for us. During the implementation of Narukom, decisions were made in order

8. FUTURE WORK

to develop a robust communication system for a RoboCup team, however it is possible to alter some implementation decisions in order to make Narukom a more suitable candidate for other distributed environments. To achieve this, a wider range of network channels should be implemented. Moreover a basic query-like language could be used to access the blackboard.

8.3 Optimization and Debugging

Every project, no matter how well it is tested and optimized during the development phase, may contain bugs and bottlenecks. Our intention and desire is to continue actively developing Narukom in order to meet the needs of as many people as possible. As a result a constant procedure of optimizing, debugging, and testing is needed to ensure that Narukom is a robust and easy-to-use framework.

Chapter 9

Conclusion

Narukom is an attempt to address a lot of communication problems our team had during the last four years in Robocup SPL. At the same time, it is a first step towards a cross-platform architecture used in robotic agents. Our journey does not end here; our Narukom has, hopefully, a long way, yet to go. The first real world test will be the Robocup 2010 in Singapore where both our inter-process, inter-robot, and team communication will be based on our framework.

9.1 Out To The Real World

Narukom is based on open-source projects, so the least we could do is to open its source in the near future. We believe that Narukom can contribute to the open-source community as a cross-platform, distributed, transparent communication framework for everyone willing to use it. I, personally, would be more than happy to assist anyone dare to use our framework. Additionally, any suggestions, comments, criticisms, improvements, and modifications are at least welcomed. By open sourcing Narukom we try to achieve two goals: (a) to encourage new coders to make available to the open-source community their code in order to advance the collaboration between programmers and (b) to protect developers from reinventing the wheel every now and then.

9. CONCLUSION

References

- [1] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., Matsubara, H.: Robocup: A challenge problem for AI. *AI Magazine* **18**(1) (1997) 73–85 5
- [2] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. *ACM Computing Surveys* **35**(2) (2003) 114–131 14
- [3] Google Inc.: Protocol Buffers. (2010) code.google.com/apis/protocolbuffers. 14
- [4] Hayes-Roth, B.: A blackboard architecture for control. *Artificial Intelligence* **26**(3) (1985) 251–321 15
- [5] Hoare, C.A.R.: Monitors: an operating system structuring concept (1974) 16
- [6] Boost.org: Boost C++ Libraries. (2010) www.boost.org. 16
- [7] Petters, S., Meriçli, T.: RoboCup Game Controller Open-Source Software. (2010) sourceforge.net/projects/robocupgc. 20
- [8] Hester, T., Quinlan, M., Stone, P.: UT Austin Villa 2008: Standing on two legs (2008) Only available online: www.cs.utexas.edu/~AustinVilla. 45
- [9] Czarnetzki, S., Hauschildt, D., Kerner, S., Urbann, O.: BreDoBrothers team report for RoboCup 2008 (2008) Only available online: www.bredobrothers.de. 45

REFERENCES

- [10] Veloso, M., Philips, M.: Gtcmunited'08: Calibrated motion, modular vision, and accurate behaviors (2008) Only available online: www.tzi.de/spl/pub/Website/Teams2008/GTCMU.pdf. 46
- [11] Ferrein, A., Steinbauer, G., McPhillips, G., Niemüller, T., Potgieter, A.: Team Zadeat - team report (2009) Only available online: www.zadeat.org. 46
- [12] Collien, D., Huynh, G.: The transition from 4legged to 2legged robot soccer. Master's thesis, University of New South Wales School of Computer Science and Engineering (2008) 46
- [13] Röfer, T., Laue, T., Müller, J., Bösche, O., Burchardt, A., Damrose, E., Gillmann, K., Graf, C., de Haas, T.J., Härtl, A., Rieskamp, A., Schreck, A., Sieverdingbeck, I., Worch, J.H.: B-Human team report and code release 2009 (2009) Only available online: www.b-human.de/download.php?file=coderelease09_doc. 46
- [14] Rofer, T., Laue, T., Weber, M., Burkhard, H.D., Jungel, M., Gohring, D., Hoffmann, J., Krause, T., Spranger, M., von Stryk, O., Brunn, R., Dassler, M., Kunz, M., Oberlies, T., Risler, M., Schwiegelshohn, U., Hebbel, M., Nistico, W., Czarnetzki, S., Kerkhof, T., Meyer, M., Rohde, C., Schmitz, B., Wachter, M., Wegner, T., Zarges, C.: GermanTeam 2005 (2005) Only available online: www.germanteam.org/GT2005.pdf. 46
- [15] Bader, M., Hofmann, A., Schreiner, D.: Austrian Kangaroos team description paper (2009) Only available online: www.austrian-kangaroos.com. 47