UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Evaluation of shared tuple spaces as communication model for semi-autonomous robots in a mobile environment

Andreas Svendby

INF-3990
Master's Thesis in Computer Science
May, 2012

ABSTRACT

Robotics can be utilized more in emergency services, and a software platform for controlling many semi-autonomous can make it more alluring to use robots in this area. Using semi-autonomous robots can allow personnel to issue robots to perform complex task, without the need of extensive training, and it allows one person to control many robots alone. One challenging aspect of using robots in emergency situations is handling communication between devices in the system. Since emergency situations can be chaotic it must be assumed that the robots have to work in a segmented network, and devices in the system must be allowed to move. Due to the mobile environment, a spatial and temporal uncoupled communication model can be beneficial; it helps handle much of the complexity acquaint with a mobile environment.

A possible architecture for robots in a semi-autonomous robot platform has been discussed, and some features for a semi-autonomous robot control unit was implemented.

A shared tuple space was designed and implemented as a communication model for use in a mobile environment. The model is based on the client-server model, consisting of three main components: hosts, clients and name servers. Hosts act as buffers in the form of a tuple space, storing tuples from clients. Hosts can be accessed remotely over a network, and clients can insert, read, and withdraw tuples from hosts, thus providing a mechanism for passing data between devices. Clients provide a interface for processes to access the shared tuple space through. The client is also responsible for combine the content from the available hosts, constructing the shared tuple space as seen from the clients point of view. Name servers simply provide device detection, letting clients locate hosts. The use of this communication model grants the ability for clients to pass messages between each other without knowing which process will retrieve it, or at time the message is retrieved.

A set of experiments were performed to evaluate the performance of the implemented shared tuple space. Performance results show that the shared tuple space has good enough performance to be used as a communication model for messages that are not dependent on low latency. It is theorized that the important aspects of

a communication model for a semi-autonomous robot platform is not low latency, but services provided by it. Messages requiring low latency should be sent over more efficient communication links, while the communication model should help remedy the complexity introduced by working in a mobile environment.

ACKNOWLEDGEMENTS

I would like to thank my advisers, Associate Professor John Markus Bjrndalen for his guidance and support during this thesis. John Markus has motivated and inspired me to keep on developing and refining the system. He has always been available and had helpful answers questions and problems.

I would like to thank my co-adviser Professor Otto J. Anshus for his support and valuable feedback during this thesis. I would also like to thank Tor Kreutzer and Brd Fjukstad for very helpful discussions and support.

I thank the technical administrative staff at the Department of Computer Science at the University of Troms for always being helpful, both during my thesis and the rest of my study. I would also like to thank the school lab at the Department of Computer Science for investing in two AR.Drones, which I have had at my disposal during this thesis.

Lastly I would like to thank my fellow students with whom I have shared office with, for all the discussions we've had.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ABBREVIATIONS

3G	3rd generation mobile telecommunications
AA	Actor Architecture
ACL	Agent Communication Language
API	Application Programming Interface
AR	Augmented Reality
ARP	Address Resolution Protocol
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
DHT	Distributed Hash Table
FIFO	First In, First Out
FIPA	Foundation for Intelligent Physical Agents
FPS	Frames Per Second
Gbit	Gigabit
GHz	Gigaherz
GIL	Global Interpreter Lock
HSV	Hue Saturation Value
IGS	Inertial Guidance System
IP	Internet Protocol
ITS	Interface Tuples Space

JS	JavaSpaces
KB	Kilobytes
KQML	Knowledge Query and Manipulation Language
LGI	Law-Governed Interaction
LIME	Linda in a Mobile Environment
MB	Megabytes
Mbit	Megabit
MEMS	Micro-Electro-Mechanical Systems
MHz	Megahertz
mm	Millimeters
NAT	Network Address Translation
NFS	Network File System
NS	Name-Server
OpenCV	Open Source Computer Vision
OS	Operating Systems
P2P	Peer-to-Peer
PID	Proportional-Integral-Derivative
POMDP	Partially Observable Markov Decision Process
RCU	Robot Control Unit
RGB	Red Green Blue
RSS	Resident Set Size
SDK	Software Development Kit
TOTA	Tuples On The Air
TS	Tuple Space
TSC	Tuple Space Client

TSH	Tuple Space Host
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus
Wi-Fi	Wireless

CHAPTER 1
INTRODUCTION

Robotics and robots have become an integral part of todays society, being used for both trivial tasks as well as more complex tasks. Robots are used in industry to perform repetitive tasks at much higher speed and precision than humans can ever do, increasing productivity and decreasing cost. But the usefulness of robots is not limited to industry; robots are used to assist in surgery[1], in warfare as Unmanned Aerial Vehicles (UAVs)[2] and bomb disposal robots[3], and more.

Even though robotics have clearly shown it usefulness, it is only used to a limited extent by emergency services. Here it is mainly limited to police forces, who may have access to robots for tasks such as bomb disposal. One area where robots could be useful for many departments is reconnaissance. Such robots could be used in various scenarios, ranging from simply getting overview of an area to participating in search and rescue missions.

One important aspect when considering whether or not to use new technology is, of course, the cost of the system. When looking at the cost of a robotics system for emergency services there are mainly two costs to consider: capital and personnel. The cost of buying and maintaining such a system is relatively low compared to its usefulness. It would be must cheaper to use a few aerial drones instead of a full sized helicopter for a certain task, assuming the drones could perform as well as the helicopter. If a robotics system could prove to be versatile and able to perform important tasks in a fulfilling matter, the cost in capital could easily be justified.

For emergency services, the personnel cost is an important aspect. Having a system where operators need extensive training just to use it both increases the cost of the system as well as its availability; its not enough to have the hardware and software available, a specially trained operator needs to be present too.

Introducing a software platform for controlling semi-autonomous robots can address the problem to some extent. Having a simple interface for issuing tasks to a robot, who can then perform the task (semi-)autonomously can limit the required training to operate such a system. This makes it feasible to have ordinary emer-

gency personnel assign robots to perform highly specialized and complex tasks. If it is easy enough to use, it can be possible for anyone in the emergency service to use it. This will increase its availability, as its not necessary to wait for a specially trained operator to arrive to use the system.

Having semi-autonomous robots can also free up personnel for other important tasks. In certain situations it might be enough to set up the system and issue tasks. If the task can be performed without any human interaction, all personnel can move on to resolve whatever emergency is at hand, turning to the system whenever they might need information from it. One example where autonomy can be very useful is to monitor a certain area. A robot can be sent out to the area to collect data (e.g. take pictures) at certain times over a long period. The data can then be processed and used to detect, for example, changes in the area.

In cases where it is possible to establish communication links to off-site locations, it might be possible to leave the control of drones to operators that are off-location, as well as giving access to collected data. This makes it possible to have experts that are off-location that can access both drones and the collected data, letting them assess the situation without being in the field.

Having a limited and easy API for the software platform and posing as general requirements to robots makes it easier to integrate various types of robots with the system. This lets personnel work in a familiar environment with any robots integrated in the system, so little to no training is required when introducing new robots. It also allows for choosing robots that fit within a certain budget and can perform the needed tasks, not forcing departments to buy expensive robots that are overly complex.

Adding such transparency on which robots are used to the system increases the burden on programmers; someone has to implement the software on the robots end, integrating it with the system and making it behave as expected. If the same task is given to two different robots, the task should be completed in a fulfilling way by both. It would be preferable if the same code could be used on two different robots to perform the same task, assuming they have so similar hardware that this is feasible. It is, of course, unrealistic that, for some arbitrary mission, the same code can be used for a helicopter and a plane.

1.1 Problem Statement

Simplifying the use of inexpensive robots and making data collected by the robots both on-site and off-site could allow such systems to be used in situations where this is impractical today. The information gathered by robots could help getting a better situational overview, and in some cases replace humans in performing hazardous tasks.

One challenge related to building a semi-automated robot system is the communication model. There are two main challenges affiliated with the communication model:

- Temporal coupling: It is common for communication models to have temporal coupling between parties who want to communicate. With temporal coupling, when a process wants to send a message to another process, the sender must be able to reach the recipient; the recipient process must be running at the time, and it must be available somewhere in the network. In many cases, the recipient must also be ready to receive a message. In a mobile environment it cannot be guaranteed that two processes on different devices are available to each other at an arbitrary time.

- Spatial coupling: In most communication models there is a spatial coupling between communicating devices. When two processes want to communicate, at least one party must know the location in the network of the other. When working in a mobile environment, devices move physically, and thus they can move in the network; the device may move in and out of different networks, changing address in the networks as it does. This makes it difficult to know the location of a specific process. If a process was locate at address *a* before, it might be located at address *b* at a later time. Sending a message to *a* can therefore cause the message to not reach its desired destination.

1.2 Scientific Contributions

This section presents the scientific contributions from this thesis.

1.2.1 Contributions

- **Shared Tuple Space**: The architecture, design, and implementation of a shared tuple space, providing a means for processes to communicate in a spatial and temporal uncoupled manner in a segmented network. A process can use a Tuple Space Client to access a shared tuple space, and the shared tuple space seen through the Tuple Space Client is dependent on the available hosts. A mechanism was developed to allow processes to detect devices in the network that are hosting a segment of the shared tuple space.

- **Experimental Results**: A set of experiments were run to measure the performance of a shared tuple space. These results are presented, evaluated, and discussed with regards to use in a mobile environment.

- **Robot Control Unit**: The architecture and design of a Robot Control Unit meant to give easy incorporation of new robots into an existing platform, along with a discussion about the use of robots in emergency situations.

1.2.2 ARTIFACTS

This artifacts were developed as part of this thesis.

- **Tuple Space Host**: Tuple Space Host is a independent segment of a shared tuple space. Each Tuple Space Host acts independently, with no communication between hosts.
- **Tuple Space Client**: Tuple Space Client is an access point to Tuple Space Hosts for processes. A Tuple Space Client is responsible for constructing a shared tuple space for a process by combining the content of available hosts.

1.3 ORGANIZATION

The remainder of this thesis is organized as follows.

Chapter 2 looks at literature related to the work presented in this thesis. In Chapter 3 the idea for the thesis is presented. Chapter 4 gives a introduction to tuple spaces; the concept of a tuple space, and some interesting examples of tuple spaces. The drone used as a prototype robot during this thesis is presented in Chapter 5. In Chapter 6 the architecture of a shared tuple space, as well as the components in the shared tuple space, and some components for a semi-autonomous robot platform, is presented. Chapter 7 presents the design of the components presented in the architecture, and Chapter 8 looks at the implementation of these components. Chapter 9 describes the methodology used for the research in this thesis. Chapter 10 describes the experiments performed to characterize the implementation of a shared tuple space, and Chapter 11 presents the experimental results, as well as a short discussion on the results. Chapter 12 discusses the work presented in this thesis, and Chapter 13 presents some concluding remarks. Chapter 14 proposes some directions for future work.

Appendix A shows the technical specifications for the AR.Drone v1.0, while Appendix B shows the technical specifications for the AR.Drone v2.0, which is set to be released in June 2012.

CHAPTER 2
RELATED WORK

2.1 OVERVIEW

The main focus of this thesis is the communication model in an unmanned vehicle system, but it also looks at the whole system, and at controlling a specific robot, the AR.Drone. Most literature on the area cover Unmanned Aircraft/Aerial Systems (UAS) and Multi-Agent Systems (MAS).

A UAS commonly consist of a single aerial vehicle, along with any sensors and payloads, communication links, ground station, and other necessary ground support equipment. As such, it does not fully cover the system discussed in this thesis. A MAS is a system consisting of multiple interacting agents, cooperating to solve a problem that may be difficult for a single agent to complete. The characteristics of an agent in a MAS varies slightly, but can be said to be[1]:

- The agent has a local view; e.g. the agent has incomplete information, or does not have capability to utilize any such knowledge
- The system does not have designated controlling agent; the system is decentralized and (semi-)autonomous
- Computation is asynchronous

Note that an agent can be a software agent, a robot, or even humans.

The difference between the system discussed in this thesis and MAS is that the system discussed here needs some sort of centralized control through agents (human operators) specifying certain condition for other agents (the drones) to perform their tasks. Also, its not as much focus on having cooperation between agents. In fact, it is desired to have agents that are able to perform tasks as independently as possible.

[1] http://aaai.org/AITopics/MultiAgentSystems, (March 28, 2012)

2.2 Autonomous and Semi-Autonomous Robot Systems

Doherty and Rudol[4] describes a system for performing search and rescue missions with UAVs. The UAVs use human body detection and geolocalization to identify injured civilians and deliver supplies to their location. They have divided a mission into two legs. In the first leg the UAVs scan an area, identifying bodies and producing a saliency map that other UAVs, or emergency services, can used. In the second leg the saliency map from the first leg is used to build a logistics plan to deliver supplies to the injured civilians. Their results from the first leg show that the UAVs were able to locate all bodies, but there were also some false positives. The geolocation is also accurate enough for delivering supplies to the detected humans. Alike the work in this thesis, their work looks at using semi-autonomous UAVs in a specific emergency scenario. Their work, however, focuses on only one scenario, and much of the work is on image processing and analysis to detect human bodies, while this thesis focuses on mostly on a more general platform, and the work discussed is mostly about the communication model.

Reed et al.[5] describe a system, SkyNET, creating a botnet by using mobile attack drones, which are controllable via auto-pilot or a mobile broadband connection (3G). The attack drone used in SkyNET is the AR.Drone with a few modifications. The drone is first piloted to an attack position. Once an attack position is found, the drone can fly to it autonomously at later points to perform the next steps. When in the attack position, the drone will start attacking networks. Once the drone gains access to a network it will start attacking loyal hosts (a loyal host is a host which consistently uses the compromised network), and lastly the drone tries to enlist the compromised host into SkyNET. Their work uses the same drone as used for this thesis, and they have implemented a system where a semi-autonomous drone is used to solve a task for an operator, much like the goal of the work in this thesis. However, the goal of their work is to reveal security issues in home networks which can be abused by such a system, while the goal of this thesis is to use such a system to aid emergency workers in different scenarios. They discuss a protocol for controlling their system, and how they attack, infect, and enlist hosts. Most of the work discussed in this thesis is on the communication model use in the platform.

Foka and Trahanias[6] looks at a robot navigating a crowded or congested environment, where it can be blocked by moving humans and objects. A common solution to such a problem is to predict the motion of obstacles and try to avoid them. The robot has to make decisions about what action to perform at each step, considering the information provided by its sensors, meaning the robot has to solve a sequential decision problem. In a completely observable environment,

the solution to such a problem is a Markov Decision Process, while if there is uncertainty or limited information, the problem is called a Partially Observable Markov Decision Process (POMDP). Foka and Trahanias proposes to integrate future motion prediction of obstacles into a global navigation model modeled by a POMDP. Their results shows that the robot is able to choose the optimal path from its location to its goal, but reject this path in favor of another path when its navigation model predicts that the robot will encounter an obstacle in its original path. Their work does not relate much to the work in this thesis, as their focus is on navigation of robots, while this thesis is about a platform for semi-autonomous robots with most of the focus on the communication model used, and no focus on such advanced navigation. However, their work would be interesting for later stages of the work discussed in this thesis.

2.3 Communication and Coordination of Autonomous Robots

Goodrich et al.[7] looks at problems related to a single human managing multiple robots. Increasing the number of robots can lead to increased workload for the operator, but allowing for adjustable and adaptive autonomy can mitigate the extra workload introduced by additional robots. They performed four experiments addressing fundamental aspects of management of a team of robots. They looked at two different management styles; sequential, where the operator gives attention to single robots, one at a time, and playbook-style, where the operator issues higher directives to clusters or sub-teams of robots. Their experiments show that high robot autonomy frees the operator to perform other tasks. It also shows that, if having adjustable autonomy, then it should also be possible to switch between the two management styles, to compensate for changes in workload. They also argue that adjustable and adaptive autonomy can improve both management styles. Their work looks at different aspects of a single operator controlling many robots, specifically how adjustable and adaptive autonomy affects it, which is important for the system discussed in this thesis. For the platform discussed in this thesis to work, it is important that a single human can control several robots. The focus is of this thesis is, however, on the platform as a whole, and specifically on the communication model used, and never reached the stage where controlling robots became a reality.

Agent communication languages (ACL) are standard languages for communication between agents. There are two proposed standards, FIPA-ACL, by Foundation for Intelligent Physical Agents (FIPA), and Knowledge Query and Manipulation Language (KQML). KQML[8] is a language and a set of protocols that

helps software agents with identifying, connecting with, and exchanging information with other software agents. KQML was originally intended as an interface to knowledge-based systems, but was repurposed as an ACL. KQML introduces a special agent class called communication facilitators. The facilitator is an agent that provide communication services such as maintaining a service names registry, forwarding messages to named services, routing messages based on content, and providing matchmaking between clients and information provides. The facilitators offers a centralized solution within ACLs.

Aszals and Herzig[9] describes the logic for peer-to-peer (P2P) communication in a MAS which aims at knowledge-sharing. It is a modal logic of knowing, saying, and asking. They do not limit searches, as some P2P networks do, so every query will get success or definite failure. The disadvantages with a P2P solution is that the location of information is unknown, while the centralized solution have the obvious disadvantage of a single point of failure. This work, as well as FIPA-ACL and KQML, is about the semantics and protocols used in communication between software agents, which relates to the main focus of this thesis, which is about the communication model between devices in the system. Their work is, however, on a much more detailed level, and focuses on easing communication between heterogeneous agents, while the work discussed in this thesis is about a software platform for semi-autonomous robots, and focuses on a general communication model.

Goldman and Zilberstein[10] looks at optimizing information exchange in decentralized cooperative MAS. They look at cooperative MASs where agents share a common goal, and agents may need to communicate to synchronize, but they might not want to share all their knowledge all the time. To optimize communication, they developed a theoretical model for decentralized control using POMDP. Their results show that a greedy meta-level approach to communication gives good results and, unlike heuristic approaches, does not need any tuning of parameters to perform well. The heuristic algorithm, set up with the worst settings, is always out-performed by the greedy algorithm. This paper discusses the communication used to allow for decentralized control of a cooperative MAS, so it looks at an alternative approach to communication between agents. One of the goals of their work is, however, to share some information, but not all, to allow agents to synchronize information, so they can continue towards their common goal. The work presented in this thesis is mainly on communication model and simply passing messages between two parties in a temporal and spatially uncoupled manner. The overall goal of this thesis is to discuss a software platform for semi-autonomous robots.

Jang et al.[11] implemented an agent system called Actor Architecture (AA), which addresses two issues within agent communication: efficient message delivery and service agent discovery. To achieve efficient message delivery the agent

naming scheme has been extended to include information about mobile agents' location, and messages to a moving agent is postponed until the agent is finished migrating. To help with service agent discovery, agents are allowed to send customized search algorithms to a tuple space, which will execute the algorithm in its own address space. The improvement seen in their system is in large part due to more efficient use of bandwidth. The work presented by Jang et al. looks at the communication between agents in a mobile environment, similar to the work in this thesis. They also utilize tuple spaces in service agent discovery. However, the work in this thesis is on a software platform for controlling semi-autonomous robots, with main focus on the communication model. Also, the communication model in this thesis is based directly on tuple spaces. Much of the focus in their work is on efficient message passing between agents that might migrate between devices. Migration of software agents is not regarded in this thesis.

2.4 TUPLE SPACE

The concept of tuple space, along with Linda, Javaspaces, and some other tuple spaces are described in detail in Chapter 4. This section will look briefly at literature and work concerning tuple spaces that is not discussed in Chapter 4.

Minsky et al.[12] describes a content-based access control mechanism for tuple spaces that conserves the Linda model. Interaction between tuple spaces and their clients is subjected to a general coordination regime called law-governed interaction (LGI). LGI allows an heterogeneous group of distributed agents to interact with each other, having confidence that an explicitly specified set of rules, called the law of the group, is complied with. Their results show that LGI can be used to add various guarantees to a tuple space while keeping the basic model of tuple spaces, preserving the attractive features. These guarantees can also be added transparently to tuple spaces, making it possible to integrate LGI with existing systems. Their work focuses on security in tuple spaces, which is an important aspect for the software platform discussed in this thesis. The communication model, which is the main focus in this thesis, is based on tuple spaces. This thesis is on a software platform for semi-autonomous robots, with most of the work focused on the communication model, and the security aspect has been disregarded, as it is something that would be addressed at a later stage.

Mamei et al.[13] describes a middleware infrastructure, Tuples On The Air (TOTA) for supporting distributed computing in dynamic networks. TOTA is composed of a P2P network where nodes can be mobile. Every node in the network runs a local instance of the TOTA middleware and holds references to a limited set of neighboring nodes. Tuples that are inserted into the system consist of two elements; content, or the tuple itself, and the propagation rule, which

defines how the tuple propagates in the network, which may include the distance the tuple should propagate and the spatial direction it should propagate in. TOTA implements fields such that tuples are able to propagate in a network in a similar manner as electromagnetic fields propagating in the air. Due to the propagation in TOTA, tuples in the system is not necessarily associated with any specific node in the network. TOTA looks at using tuple spaces in a distributed, mobile environment, alike the main focus of this thesis. It differs from the work in this thesis by that it is meant as a middleware for propagating tuples over a network, while the work presented is meant as a communication model between nodes in the system, storing tuples at hosts until they are retrieved.

CHAPTER 3
SOFTWARE PLATFORM

The software platform is aimed at providing a system for controlling semi-autonomous robots for use by emergency services and other non-military organizations. The system should try to fulfill these points:

- Provide an easy way to issue commands to robots
- Robots should be able to perform fairly complex tasks
- Support at least tens of robots at a time
- Support computation resources
- Support several operators

A task is a unit of work that is issued to a device in the system, e.g. a robot. It might vary from simply moving to a certain location, to following a set of waypoints and doing something specific at each waypoint. A job is a fragment of a task; a task can consist of many jobs. If a task is to decode a video stream, a job is to decode a single frame from the video stream. A mission (e.g. to search a certain area for people) can be divided into several tasks (e.g. searching a smaller portion of the area). In this thesis we will keep to missions consisting of a single task and thus mainly discuss tasks, unless specified otherwise.

Handling of operators can be done in many different ways. By default all operators should have equal access to all robots, although in some cases access to a specific robot should be limited. If one operator is directly in control of a robot, all other operators should have limited access (e.g. they should not be able to issue any control-commands to it). It might also be desirable to have priority of operators; some operators have more access than others, maybe even allow some operators to override others.

In an emergency situation it might be useful to have many robots available, but it might be a limited amount of personnel available to operate the system. It is therefor desired that any amount of robots may be controlled by as few operators

as possible, maybe even a single operator. This is why it is important to have semi-autonomous robots; the robots should be able to perform as much of a given task without human guidance. If each robot required its own operator working specifically with it, it would not be feasible to utilize many robots at once.

As it is normal to have a staging area in emergency situations, it can be assumed that it is possible to have at least one static operator. Of course robots are assumed to be mobile, in addition to some operators, and maybe even some third party resources. This poses strict requirements to the underlying system when it comes to handling mobility. First of all, it cannot be assumed that an existing network is available, so it might be necessary to set up an ad hoc network. Second, it should be assumed that the mobile parties in the system will have limited availability, which must be handled.

It might be possible to use the 3G-net, which allows for high availability, but even this is not good enough. It should not be assumed that it is always possible to have access to 3G, as there might not be a signal everywhere, or the network might be overloaded. There might also be situations where the the 3G-network is down. Due to this is should still be possible to set up a local network.

The mobile nature of the system poses some problems concerning the communication model, specifically concerning spatial and temporal coupling of communication. If a process wants to send a message to another process, the recipient process must commonly be available for the sender to pass a message to it. Imagine having a urgent message to process on a mobile device. Using communication models with temporal coupling, the sender process must simply try to the send the message again and again, until the recipient is available and ready to receive the message. Communication will be a lot simpler to handle if the recipient does not need to be available for a message to be sent. This way the sender can just send a message, and the recipient can retrieve it when possible. This problem can be solved by using a communication model with temporal uncoupling of sender and recipient.

Another problem in a mobile environment is related to spatial coupling. It is natural that the IP address of devices may change in a mobile environment, so having the possibility to address messages to devices in another way would be desirable. This can be achieved by spatial uncoupling. Even though sender and recipient both can stay oblivious to the other part, they both need to know about a global buffer where they can store and retrieve messages.

CHAPTER 4
TUPLE SPACE

In computer science a tuple is an ordered set of values, commonly separated by a comma. Typical uses for tuples are passing parameters in function calls or representing a set of value attributes in a relational database. In Python a tuple is very much like a list, the main difference being that it is immutable once it has been created. Thus tuples are useful for data which should not be altered.

The term tuple space (TS) was first introduced in the programming language Linda[14] as a generative communication model. A TS can can be explained simply as a collection of tuples, or more specifically as a shared associative memory used for communication between processes. In its most basic form a TS provides three operations; *out* inserts a tuple to the TS, *in* withdraws a tuple from the TS, and *read* reads a tuple from the TS, but without withdrawing it. Extraction of a tuple from a TS is done by matching the components of the tuple. Each component may be either an actual parameter or a formal parameter, allowing matching on data type or specifically on content.

Using a TS as communication model, two processes A and B can communicate by inserting and withdrawing tuples from the TS. If a process wants to send data to another process, it generates tuples and inserts them into the TS, while a process that wants to receive data reads or withdraws them from the TS. Gelernter calls this model generative because:

> ”(...) until it is explicitly withdrawn, the tuple generated by A has an independent existence in TS. A tuple in TS is equally accessible to all processes within TS, but is bound to none.” [14]

He specifies two fundamental characteristics in [14]: communication orthogonality and free naming.

Ordinarily, when sending messages between two parties the sender of the message must somehow name the recipient of the message specifically. The recipient, however, does not need to know anything about the sender, and can receive messages from any sender. With orthogonal communication the sender has no

knowledge of who receives the message, just as the recipient does not know who sent it. This gives spatial uncoupling.

Since tuples exist independently in a TS after insertion, a process may insert a tuple into a TS and then terminate. At some later time another process can retrieve this tuple from the TS. The process retrieving the tuple may even start after the tuple was inserted. This gives temporal uncoupling.

Note that all examples in this chapter is based on the original examples used in [14].

4.1 LINDA

Linda is a distributed programming language using a model for concurrent programming called generative communication. Generative communication lets Linda be fully distributed in space and distributed in time, separating it from most other distributed languages. In [14] Gelernter introduced tuples space as the basic communication model in Linda:

> ”The abstract computation environment called ”tuple space” is the basis of Linda’s model of communication. An executing Linda program is regarded as occupying an environment called ”tuple space” or TS.”

The following sections will describe the basics for the operations in Linda. First the operations are defined in their simplest form, before looking at structured naming.

4.1.1 DEFINITION OF OPERATIONS

In their simplest forms, tuples used in the out, in, and read statements consists of an actual parameter N and a list of parameters P_2,...,P_j. Each of the parameters P_2,...,P_j may be either an actual or a formal. Formal parameters are parameters as specified when defining a function, while actual parameters are the arguments used when calling the function. In the following, a formal parameter is when only the datatype is specified, while an actual is when a specific value is provided. The statements will then be the following:

out(N, P_2,...,P_j)
in(N, P_2,...,P_j) (taken from [14])
read(N, P_2,...,P_j)

Assuming that an out statement is executed with only actual parameters. The tuple N, P_2,...,P_j would be inserted into the TS and the executing process would

continue immediately. For the in statement, assume that P_2,...,P_j are all formal parameters. If a tuple exist in the TS whose first component matches N, and the data type of all other components match their corresponding components in the request, then the the tuple is withdrawn. When the tuple is withdrawn from the TS, the values of its actuals are assigned to the formals in the in statement.

The only difference between the read statement and the in statement is that, in a read statement, any matching tuple remains in the TS after execution. Both these statements are blocking, meaning that if either of the two do not find a matching tuple, they suspend until one is available and then proceeds.

Two in statements trying to withdraw the same tuple cannot both get it. One should get it while the other should be suspended. This makes it necessary to have atomic insertion and withdrawal from the TS. Note that the order that statements are served by a TS is arbitrary, so the result of a read statement and a in statement contending for a tuple is non-determined. If the read statement is server first, the tuple will still be there for in to withdraw. However, if switched, the tuple will not be there when read is served, and it will be suspended.

4.1.2 Structured Naming

When withdrawing or reading a tuple from a TS, the parameters P_2,...,P_j does not need to be formals; each of the parameters can be actuals. Actual parameters in a tuple withdrawn or read from a TS, including the name-value actual N, forms a structured name. A statement

in(P, i:integer, j:boolean) (taken from [14])

requests a tuple with the name ”P”. Here both the second and third parameters are formals, but the statements

in(P, 2, j:boolean)
in(P, i:integer, FALSE) (taken from [14])
in(P, 2, FALSE)

are also allowed, where these either one, or both, parameters are actuals. Here tuples with the structured name ”P,2”, ”P,,FALSE”, and ”P,2,FALSE” respectively are requested. For a tuple t in TS to match a request, all actual components in the request must be identical to their corresponding components in t.

When inserting a tuple into a TS, each component of the tuple being inserted, except the name-valued actual N, can be formals. This gives the out statement a form of inverse structured naming. So both

out(P, 2, FALSE) (taken from [14])

and

out(P, i:integer, FALSE) (taken from [14])

are allowed out statements. The tuple that is inserted into the TS from the second statement can be received by any request that specifies the first component as ”P”, the last component as ”FALSE”, and the second component as some *actual* type of integer. It is important to note that formals can never match formals. If a tuple in a TS has a formal, the corresponding component in a request must be an actual of the same data type.

4.2 LINDA IN A MOBILE ENVIRONMENT

Linda in a Mobile Environment[15], or LIME, is an extension to Linda to deal with a mobile environment. Two types of mobility is taken into consideration: physical mobility and logical mobility.

Physical mobility is simply the movement of physical devices in the system, such as laptops of mobile phones. Logical mobility is the movement of mobile agents, which is processes able to migrate between hosts without any loss of code or state. The problem with Linda in a mobile environment is that it does not provide any good solution to disconnections; disconnections are just an inconvenience that should not happen. In contrast, a mobile environment assumes disconnections to be a common event.

LIME uses a transiently shared TS for coordination between mobile components. The underlying system is explained in [15] as:

> ”(...) mobile agents are programs that can travel among mobile hosts. They are ”active” components of the system. Mobile hosts are roaming containers for agents, to which they provide connectivity.”

4.2.1 INTERFACE TUPLE SPACE

Access to a TS in LIME is through an interface tuple space (ITS), as seen in Figure 4.1. The TS seen through an ITS is a transiently shared TS. The ITS provide the same interface as Linda, and it follows the mobile agent if it migrates to another host. To handle movement the content in an ITS appears as the content of all co-located mobile agents merged into one. So if two mobile agents are accessible to each other, they can access the content of each others ITS. However, if they lose connection to each other, they also lose access to the content of each others ITS. In a transiently shared TS, the re-computation to determine what content is

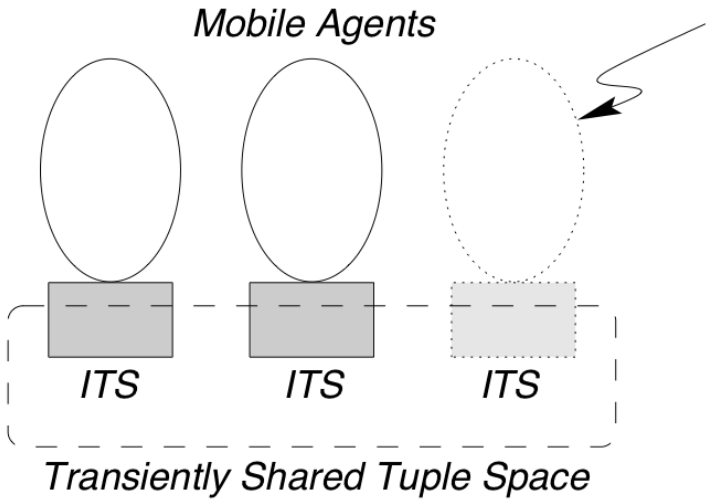

Figure 4.1: Transiently shared tuple space in LIME. The circles are mobile agents, and the arrow indicates that a mobile agent migrating to that location. The figure is taken from [15].

available in the TS is done on arrival or departure of mobile agents. Note that a tuple inserted by a process A is, by default, inserted into As ITS. The location of a tuple can be specified, as discussed later.

LIME supports multiple ITSs and private TSs on mobile agents. This allows for two mobile agents to share one or several TSs, while not sharing other TSs. In addition, all mobile agents have an ITS called LimeSystems, which does not allow withdrawal of tuples, and holds information about the mobile agent. Note that a single host can contain several mobile agents. Any mobile agent located on a single host can share a TS, and they will always be connected. A TS shared between agents on a single host is called a host-level TS. When hosts are connected to each other over some sort of network, they merge their host-level TSs into a federated tuple space. The content of a federated TS is thus transiently shared over the network.

One problem introduced by the mobile environment is to retain the time uncoupling as seen in Linda, as described in [15]. In Linda a process A may insert a tuple t into a TS, and then terminate. Another process B may then, at some later time, retrieve the tuple t. A similar case may happen in LIME. Process A and process B are co-located, so the content of their ITSs are available to each other. Process A may insert a tuple t that is intended for process B. Before process B retrieves the tuple, A departs, making the content of its ITS unavailable for B.

Thus when B tries to retrieve *t*, it is no longer available.

To solve this problem LIME lets the out statement provide a intended location. This means that process A can specify that the tuple *t* should be inserted into process Bs ITS. The tuple is then first inserted into As ITS, and then moved to Bs ITS. If B is not available, the migration is suspended until B becomes available again. The tuple will still be available and can be retrieved before it can be migrated to Bs ITS. Similarly, it is possible to both specify current location and intended location of a tuple when reading or withdrawing a tuple.

4.3 JAVASPACES

JavaSpaces (JS)[16] is, as the name suggests, an implementation of a TS in java. The main difference between JS and conventional TSs is that JS does not contain tuples, it contains instances of an *Entry* class. In JS, each separate space is an object store, making JS a hybrid between an object store and a TS. The API for JS is similar to that of Linda. The API provides three basic operations, write, read, and take. These corresponds to out, read, and in, respectively, in Linda. Although JS uses objects, they cannot be modified while in a JS; they must be removed, updated and reinserted for any change to take place. A transaction service is used to secure atomicity of operations on a space. Both a single operation on a single space, and multiple operations on multiple spaces are supported in the transactional service. In addition to the three basic operations, two more operations are supported: notify and snapshot. Notify takes an object and a template and asks the space to notify the object whenever an object matching the template is added to the space. Snapshot takes an entry and returns another entry object that contains a snapshot of the original entry. Any modifications to the original entry will not affect the snapshot.

A space in JS handles concurrent access itself, and provides reliable storage. A stored object can have a renewable lease time, so it is removed after a certain amount of time (or if it is explicitly withdrawn), unless the lease is renewed. If there is no lease, the object will remain in JS until it is explicitly removed. Lookup in JS works similar to that in conventional TS; a template is created, where its fields can be set to a specific value or left as null, in which case it acts as a wildcard. The template matches an object in JS if all its specific fields is an exact match to its corresponding fields in the object.

As mentioned above, objects in a space cannot be modified; they are just passive data. But when retrieving an object from a space, either by take or read, the local copy of the object can be both modified and its methods invoked, even objects that have never been seen before.

4.4 LINUXTUPLES

Linuxtuples[17] is an open-source implementation of a TS server, designed to run on a cluster of computers with x86 processors and Red Hat Linux 8.0, but it works in most Unix environments. The TS is running and maintained from one machine on the network. It is implemented in C, but provides an API for both C and Python. The API in C and Python supports the same operations, so this section only looks at the Python API.

4.4.1 LINUXTUPLES API

The API is similar to that of Linda, with some additional operations. The three basic operations put, get and read corresponds to out, in and read, respectively, in Linda. In addition to these, the operations get_nonblocking, read_nonblocking, dump, count, and log are provided.

The two first are, as the name suggests, non-blocking versions of the two operations get and read; they will return immediately even if no matching tuple was found. The dump statement returns the contents of the TS, while count returns the number of tuples in the TS. The last operation, log, prints a running log of the activity on the TS server to stdout. Linuxtuples does not support formal parameters as in Linda. Components can either be actuals or wildcards (as in JS), where wildcards are set using *None*. There is also no requirement for any parameter to be a name tag.

The put statement works almost identical to out; it takes a tuple and inserts it into TS. However, unlike Linda, put does not support formals or wildcards. The tuple must consists solely of actuals. The get and read statements, in addition to their respective non-blocking versions, takes a template as input. The template can consist of actuals or wildcards. Wildcard components accept any type of variable. The statement

```
get(”Name”, 2, None)
```

may return the tuple (”Name”, 2, False), as well as the tuple (”Name”, 2, 4), which one is non-determined. The non-blocking versions of get and read will return immediately and, if no matching tuple was found, None is returned.

The dump statement returns a list of all the tuples in the TS. A list of templates may be given, in which case only tuples matching at least one of the templates will be returned. The count statement works identically to dump, but returns the number of tuples.

In addition to the log statement, which prints a running log of TS activity, a Python script for controlling jobs is provided. The script supports functions such

as showing a continuous log of the TS activity, show all current tuples in the TS, show size of the TS, in addition to functions specifically for running jobs on a cluster.

4.4.2 SIMPLETS

A simple TS implementation was also written in Python loosely based on SimpleTS[1] and heavily based on Lindypy[2]. This implementation adopts the API of Linuxtuples, supporting non-blocking versions of read and get, put, count, and dump. See Section 8.2.1 for more on the implementation.

[1] http://www.cs.uit.no/ johnm/code/teaching (March 27, 2012)

[2] https://bitbucket.org/rfc1437/lindypy/src/2c0576344f5f/lindypy/TupleSpace.py (March 27, 2012)

CHAPTER 5
AR.DRONE

The Parrot Augmented Reality Drone(AR.Drone)[18] is a radio controlled quadrotor, often called quadrocopter, which is designed and built by Parrot[1]. Originally the AR.Drone was contolled by iPhone, iPod touch, or iPad, but due to its popularity mobile applications have been created for Android devices, as well as other mobile devices. A software development kit (SDK) has been released[19], which have spurred the creation of various applications for the AR.Drone on many different platforms. The official API for controlling the AR.Drone is in C, but unofficial APIs for various programming languages have been created.

The AR.Drone have four electric motors, an inertial guidance system (IGS) which consist of an accelerometer and two gyrometers, an ultrasonic altimeter with a range of 6 meters, and two cameras, one pointing forward and one downwards. The communication link between the AR.Drone and the controlling unit is over an ad hoc wireless (Wi-Fi) network established by the AR.Drone itself. For more information about the technical specifications see Appendix A.

Due to the low cost of the drone, the open source API, and the commodity communication link, the AR.Drone has become popular as a prototype platform within research[20, 21, 5, 22, 23]. As the AR.Drone 2.0 is set to be available from June 2012, it is likely that the AR.Drones popularity within research will continue, as it is set to have additional and more accurate sensors, as well as improvements to the video stream. The technical specifications for the AR.Drone 2.0 can be seen in Appendix B.

[1] http://www.parrot.com/usa/, (April 13, 2012)

Figure 5.1: The Parrot AR.Drone. The image was taken from the official AR.Drone webpage[2].

[2]http://ardrone.parrot.com/parrot-ar-drone/usa/ (April 17, 2012)

CHAPTER 6
ARCHITECTURE

6.1 OVERVIEW

Figure 6.1 shows the architecture of the software platform. The system can consist of three different types of components: user clients, workers, and compute resources. The basis for the system is tuple spaces, which describe the interaction between devices in the system. The concept of a tuple space is described in detail in Chapter 4. Each device in the system *can* host a part of a shared TS through a tuple space host (TSH), but it is not required. The TS is shared, transitory, and accessible from a tuple space client (TSC). The TS seen from a TSC is shared over one or many TSHs, and its content will vary depending on the availability of the TSHs. A TSC will only see *one* TS; there cannot exist two shared TSs in the same network. In the rest of this thesis, a reference to *the TS* or *the shared TS* means the shared TS on the network in question.

A name server (NS) can be contacted to locate TSHs. NSs hold information about all TSHs and all other NSs. NSs can either be static (run on a static computer), or dynamically created on TSHs. Static NSs are required to access resources outside the subnetwork. Dynamically created NSs are required so devices can find TSHs when in a limited network.

As the system is based on the communication model provided by tuple spaces most of the communication going on in the system passes through a shared TS. In certain cases where it is important to have low latency, it might not be feasible to use a communication model such as TS, which has more complexity, and thus more overhead, when passing messages, compared to sockets. In these cases a direct link is set up.

Communication between devices in the system must go through some sort of link. As discussed in Chapter 3 it cannot be assumed that a existing network can be used. To get a reliable network the communication link used should have long range and enough bandwidth to transfer at least one video stream along with some other data. However, in this thesis, WiFi is used along with an existing network as

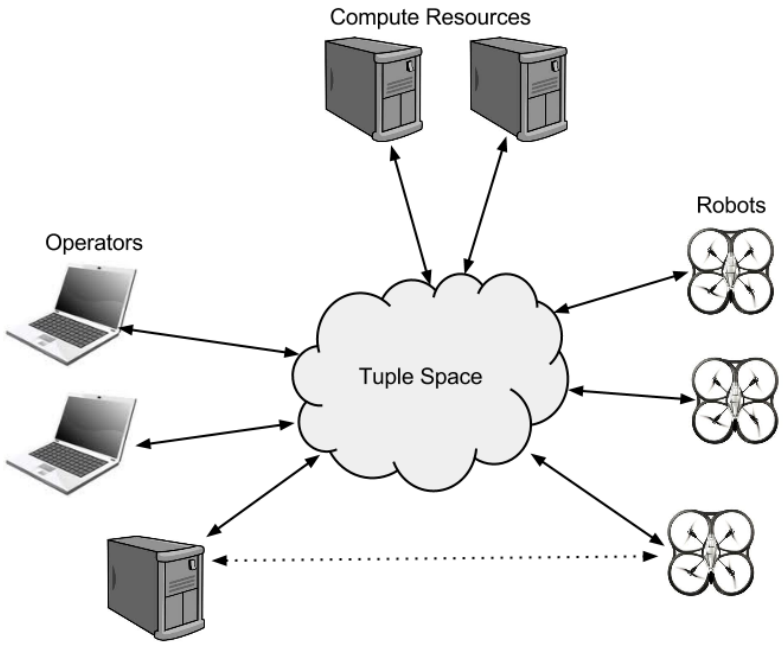

Figure 6.1: Architecture of software platform.

communication link. In the next sections we will look at the architecture of tuple space hosts, name servers, tuple space clients, robots, and a compute resource running on a cluster.

6.2 Tuple Space Hosts

A TSH consist of a locally running TS and a TS server, as seen in Figure 6.2.

The TS server is the access point to the local TS of a TSH for TSCs. TSCs contact the server, which performs a lookup in the local TS, and the result is returned to the requesting TSC. NSs are used to enable TSCs to locate TSHs, as described in Section 6.3. The TS server contacts NSs to proclaim that they are running a local TS that is part of the shared TS. There is no communication between TSHs. Each TSH is oblivious to the existence of any other TSHs. A device is limited to only hosting a single TSH at a time.

6.3 Name Server

Name servers are used to enable TSCs to locate TSHs. Figure 6.3 shows the NS architecture. A NS holds information about TSHs and other NSs. The system is

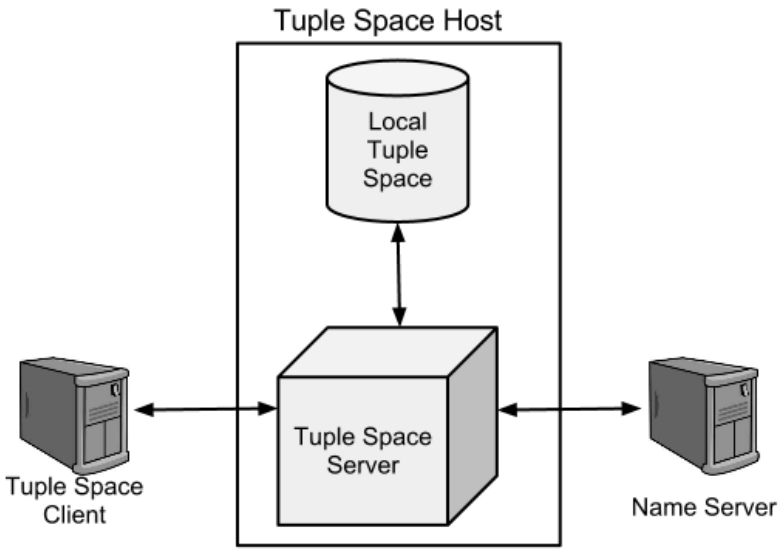

Figure 6.2: Architecture model of Tuple Space Host.

designed to potentially contain several NSs. It is especially essential that each part in a segmented network contains a NS. There are two types of NSs: dynamic and static.

Dynamic NSs broadcast their existence to each other over multicast as well as a standard point to point communication link, while static NSs communicate only over a point to point communication link. A dynamic NS can be found either through other NSs or by listening for its multicast message. A dynamic NS can only be created by a TSH. When a TSC needs to access the shared TS, it first contacts a NS, which returns a list of all known TSHs and all known NSs. No guarantee can be given that these lists contain all the TSHs and NSs, they are only the ones that are known by this NS. It can neither be assumed that these are available to the requester.

6.4 Tuple Space Client

The TSC acts as the access point to a shared TS for devices. When the TSC is to access the TS, it first contacts a NS to receive a list of TSHs. After receiving a list of TSHs, the TSC contacts each TSH until its request is fulfilled. Fulfillment of a request will be discussed in Section 7.4. There is no limitation on the number of TSCs per process or per device; a single process can host many TSCs, and a single device can have many processes, each with one, or many, TSCs.

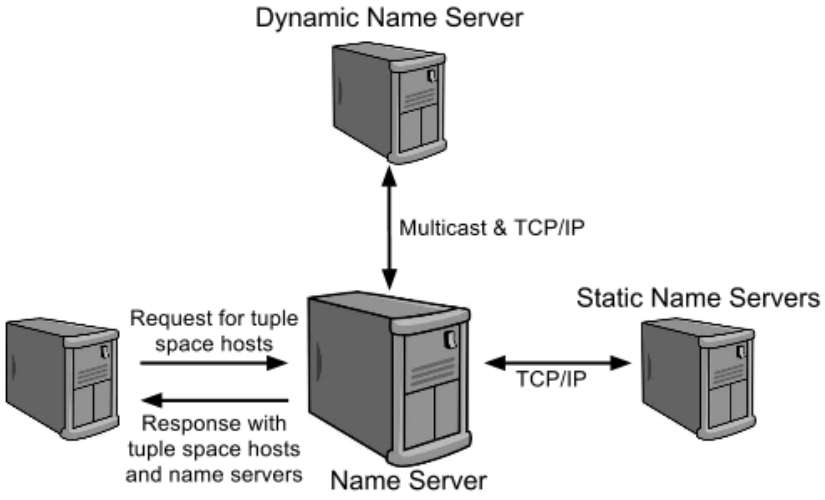

Figure 6.3: Architecture of name servers.

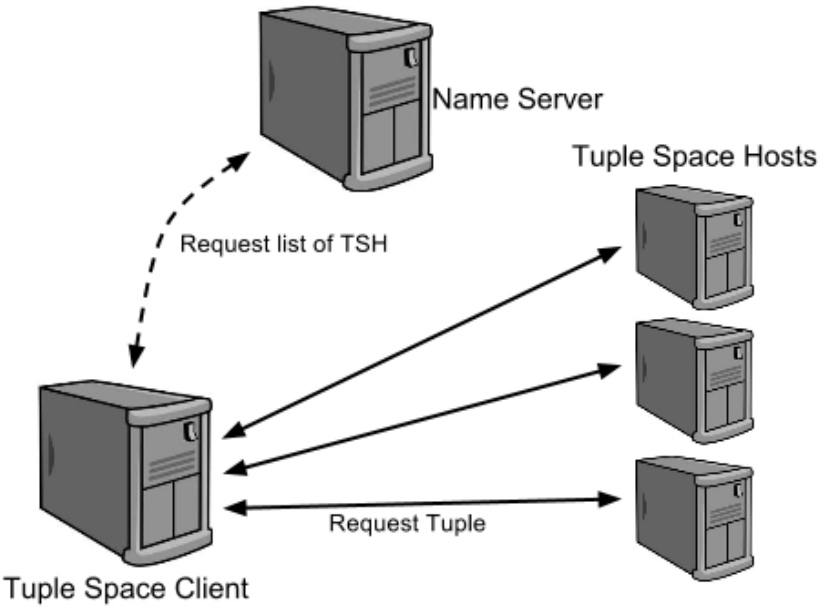

Figure 6.4: Architecture model of Tuple Space Client.

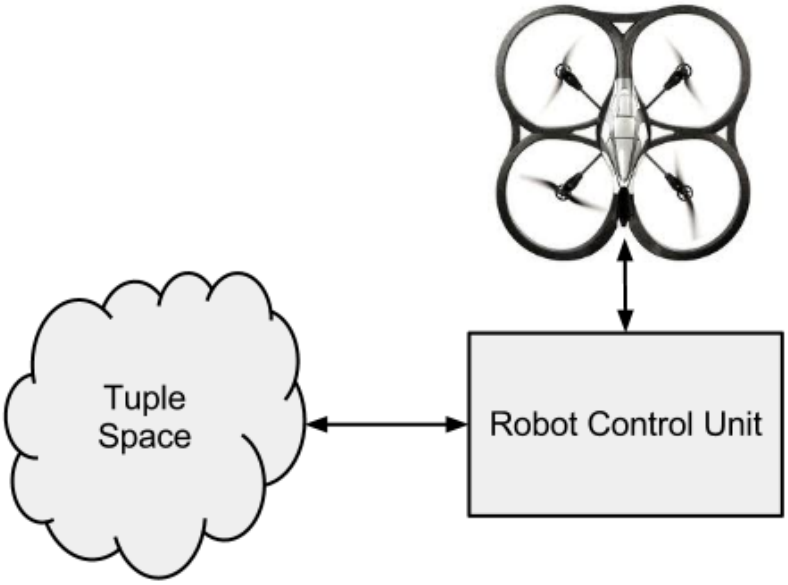

Figure 6.5: Architecture model of robots.

6.5 Robot Control Unit

Robots are specified by their class, brand and model; the class is the type of robot (plane, helicopter, quadrocopter), brand is the type of class (e.g. AR.Drone), while model is the type of brand (AR.Drone 1.0, AR.Drone 2.0).

A robot, as discussed in this thesis, actually consist of two components; the robot itself and the Robot Control Unit (RCU). The RCU is simply a computer of some sort, which receives data from the robot and any other sensors available, and control the robot based on this data and input from operators. The RCU consists of a main controller and a robot controller. The main controller is dependent on the robots class and the available sensors; e.g. a plane and a helicopter work in fairly different ways, and extra sensors need to be handled. The robot controller is dependent on the brand and model of the robot; two different helicopters can possibly perform the same tasks, but their interface will probably differ a lot.

The RCU is also the component that communicates with other devices through a shared TS. It will receive messages which are handled by the main controller. The RCU can also run a local TS, that is not a part of the shared TS. It is also possible to host a part of the shared TS, but it is not recommended as it should be assumed that the robot will often be disconnected from the rest of the system.

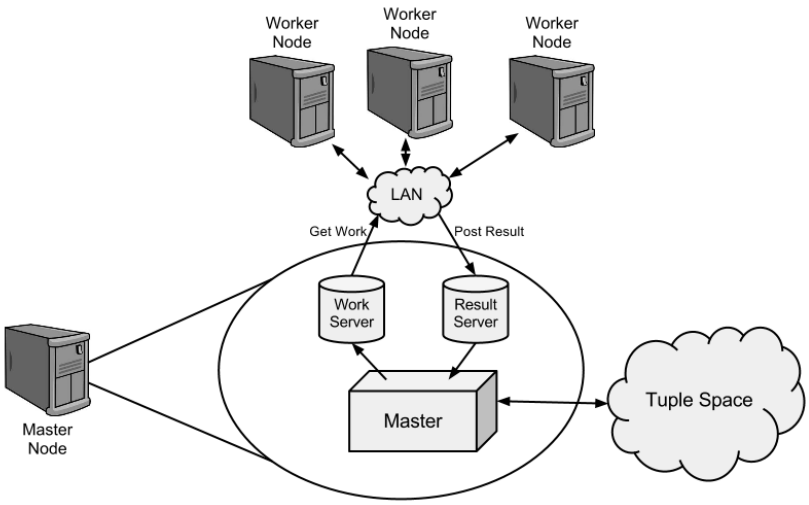

Figure 6.6: Architecture model of a compute resource running on a cluster.

6.6 Compute Resource on a Cluster

A simple compute resource was created to run on a cluster of computers for experimental reasons. This section will describe its architecture, but it's important to note that this compute resource does not present any definite structure of a general compute resource.

The compute resource consists of a single master node and many worker nodes. The master retrieves tasks from the shared TS. A task consists of many jobs, which are also retrieved from the shared TS, and the task specifies how to perform the jobs.

Once a task is received, the master retrieves jobs. The jobs are preprocesses as specified by the task, and inserted into a work-pool. Workers will retrieve work from the work-pool and perform it as specified by the task. Results are inserted into a result-pool and later retrieved by the master. Results are combined by the master into a final result that is inserted into a shared TS.

No logic on how to preprocess jobs, perform work, and combine results is known to neither the master nor workers by default. This logic is provided by the initial task.

CHAPTER 7
DESIGN

7.1 OVERVIEW

In this chapter the design of key components in the software platform will be described. First the design of tuple space hosts will be presented, before looking at the design of name server, tuple space clients, robots, and compute resources. Lastly, CSP is mentioned briefly.

7.2 TUPLE SPACE HOSTS

The TS server is responsible for handling all remote requests to the local TS. A remote client sends a request over TCP/IP, and each request is handled by a separate thread. The threads then use the TSs access logic to access the the TS and process the request. The response is sent back to the thread, which returns the result to the remote client over the same TCP/IP link that the request was sent over.

A local TS is comprised of three components; the actual tuple space, the tuple space default access logic, and the tuple space specific access logic. The default access logic contains default methods to access the TS. The specific access logic contains logic related to the TS that is specific to the TS implementation, such as methods to start the TS and terminate the TS, and, if necessary, methods overriding the default methods for access to the TS. The default access logic handles cases where a function that the TS does not support is called, making sure a result is always returned. It also protects the methods for accessing the TS with a synchronization primitive, thus preventing concurrent access to the local TS.

There are some requirements for TS implementations. A TS must support put, and polling, non-blocking versions of read and get, that always returns a result. Operations to find all tuples matching a template must also be supported. An API in the used programming language must also be available for the access logic to

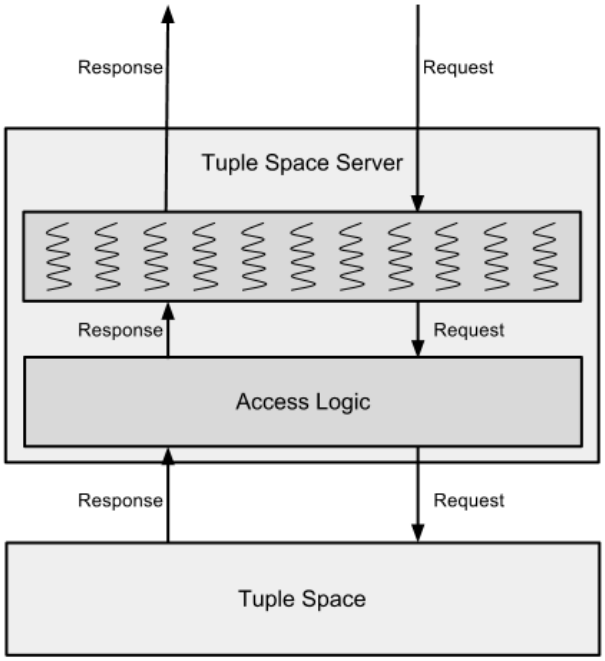

Figure 7.1: Tuple Space Host design.

use. Of course, if it is not provided by the TS, a wrapper can be written.

It is important that the TSH use non-blocking, polling, versions of get and read on the local TS. If they are blocking, a request from a client will stop at the first TSH contacted and not propagate to other TSHs, if needed. Using blocking operations would cause the request to either be successful and return a result from the first TSH, or it will block until the first TSH can return a matching tuple.

All tuples added to the TS is timestamped by the default access logic, and when returning tuples, the timestamp is used to decide how long it has been in the TS. The returned value to a TSC contains thus contains two elements; the first is the element itself, the second is the time spent in the TS. As some TS implementation may allow duplicate tuples, a counter must be used to keep track of how many duplicates are left of that tuple. All duplicates share timestamp, which is the time that last duplicate was inserted into the TS.

When the TSH receives a request, it will process the request and send the result back to the TSC. Once the result is sent, the TSH will wait for acknowledgement confirming that the result was received. If the acknowledgement is not received within a certain time limit the TSH will not consider it as a successful transmission. Only operations that withdraw tuples are affected by the lack of an acknowledgement message; any withdrawn tuples will be re-inserted into the TS. Tuples inserted into the TS will not be removed as a result of missing acknowl-

edgement.

A TSH listens to heartbeats from NSs via multicast, and pings static NSs to check their state. If there are too few NSs up, a dynamic NS is spawned.

7.3 Name Server

Name servers are used to enable devices to locate TSHs. NSs use the TCP/IP protocol for communication link and support multithreading. A NS maintain information about TSHs and NS, but it should not be assumed that an arbitrary NS knows about all TSHs and NSs. Information about the two types of services are maintained separately. A NS provides functions to add and remove TSHs and NS (separate functions for the two), request IP address of TSHs, request IP address of NSs, and ping the server. Note that the ping functionality is specifically to determine if the server is running.

Both TSHs and NSs must send a heartbeat message at certain intervals to verify that they are still running. A TSH only need to send a heartbeat to one NS, as all messages from TSHs are relayed to other NSs. TSHs send the same message for heartbeat as they do to add the TSH initially. A static NS only communicate over TCP/IP, while dynamic NSs use both TCP/IP and multicast. A dynamic NS broadcasts its existence over multicast to a specific domain. It will also listen to heartbeats on that multicast domain, as well as receiving heartbeats via TCP/IP.

Each IP address (TSH or NS) is associated with a timestamp, which is the time of the last heartbeat received. The NS checks the maintained information at intervals, removing any that is deemed too old.

7.4 Tuple Space Client

The TSC is responsible for providing an access point to a shared TS for processes and threads. The TSC consist of two components; a TS API, and access logic to TSHs. The TS API is a set of methods defining the interface of the TSC, and thus the shared TS, while the access logic is used by these methods to pass requests from the TSC to TSHs.

The TSC API provides a similar interface as the local TS; it contains the methods put, get, read, dump, and count. The get and read methods are polling, non-blocking methods, but blocking versions are also provided. When inserting a tuple, the tuple is inserted into a FIFO queue and stored until it has been successfully inserted into the TS. A thread will retrieve tuples from the queue and try to insert them into the shared TS.

The access logic contains methods to locate TSHs, and to send requests to,

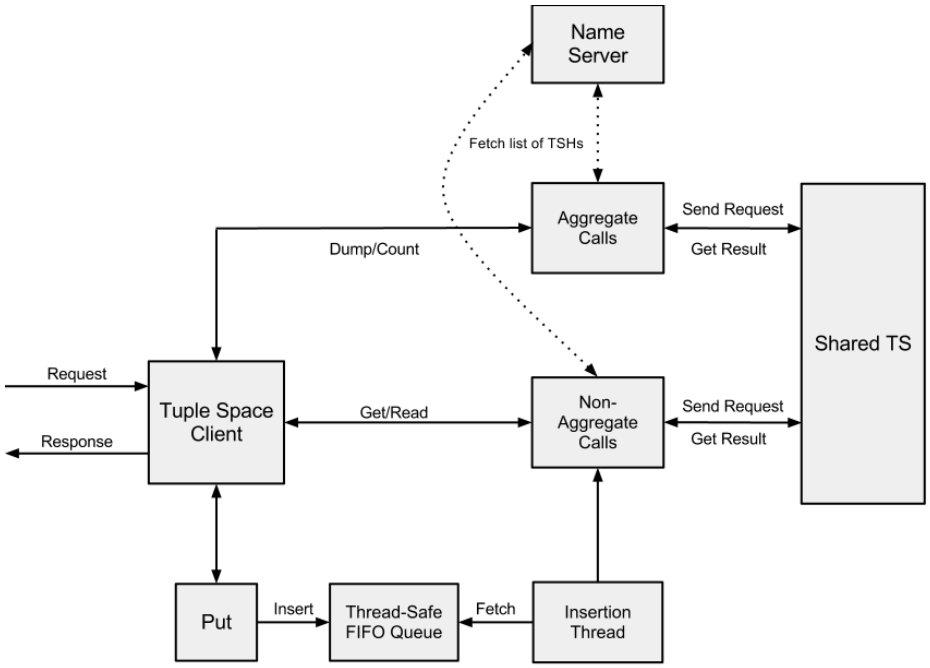

Figure 7.2: Tuple Space Client design.

and receive results from, TSHs. Two different methods are used to access TSHs, one for requests that only need a single result, and one for requests that need to aggregate results from all TSHs. The access logic also maintain a list of TSHs and NSs.

After a request is finished, the access logic will keep the current connection to a TSH open for some time. When a new request is to be sent and a connection is still open, this connection is prioritized when sending the request. If another TSH has to be contacted, any locally running TSH is prioritized. Even though it is hosted locally, access to a local TSH still goes through the TS server using TCP/IP. When accessing the TS, if the list of TSHs isn't too old, it is used to contact the TS. If it is too old, a NS is contacted and a new list is retrieved. When contacting a NS, the list of NSs is simply iterated, and the first response is used. If no NS responds, multicast is used to locate a NS.

As it is not possible to have blocking get or read on the TSHs, they must be implemented at the TSC. A blocking get or read simply call their corresponding non-blocking versions at increasing intervals, returning when a matching tuple is found. To prevent spamming the shared TS with requests, the interval is initially low, but increase for each time the request has to be repeated, going up to a maximum limit.

When contacting a TSH, the TSC first sends the request, then waits for a result.

After a result is received an acknowledgement is sent to the TSH to confirm that the result was received. If the communication link fails before a result is received, the request is regarded as failed, and the TSC contacts the next TSH. If a result was received the request is considered successful, even if the acknowledgement was not successfully sent. Some requests need to aggregate results from all TSHs, in which case the request is fulfilled when the TSC has tried to contact all TSHs. The TSC does not have to successfully contact every TSH to fulfill the request. The request will be fulfilled even if none of the TSHs could be contacted. The acknowledgement message is used by the TSH to handle disconnections.

For the non-blocking versions of get and read, the request is fulfilled when a TSH returns a matching tuple, or when all TSHs have been contacted. Again, it is not necessary to successfully contact a single TSH to fulfill the request. For the process using a TSC, the put operation will always be fulfilled right away, but the TSC will not necessarily insert the tuple into the TS right away. The TSC will insert the tuple when its the next in the queue and a TSH is available. Blocking read and get will be fulfilled once a matching tuple is returned from a TSH.

7.5 Robot Control Unit

The robot control system running on the RCU is divided into two components: a main controller and a robot controller. The main controller handles communication with the shared TS and passes retrieved tasks to the robot controller for processing.

7.5.1 Main Controller

The main controller holds the robot controller, a TSC, and possibly a local TS. It retrieves messages from the shared TS and broadcasts the existence of the robot via the shared TS. Messages retrieved are processed, and if it is a task meant for the robot the task is passed on to the robot controller for processing.

7.5.2 Robot Controller

The robot controller consist of a universal controller and a specific controller. The universal controller handles the logic that all robots must contain, while the specific controller handles logic that is dependent on the brand and model of the robot. The controllers are tightly coupled; they are both dependent on the other. The universal controller handles execution of tasks, but the logic on how to execute tasks must be defined in the specific controller.

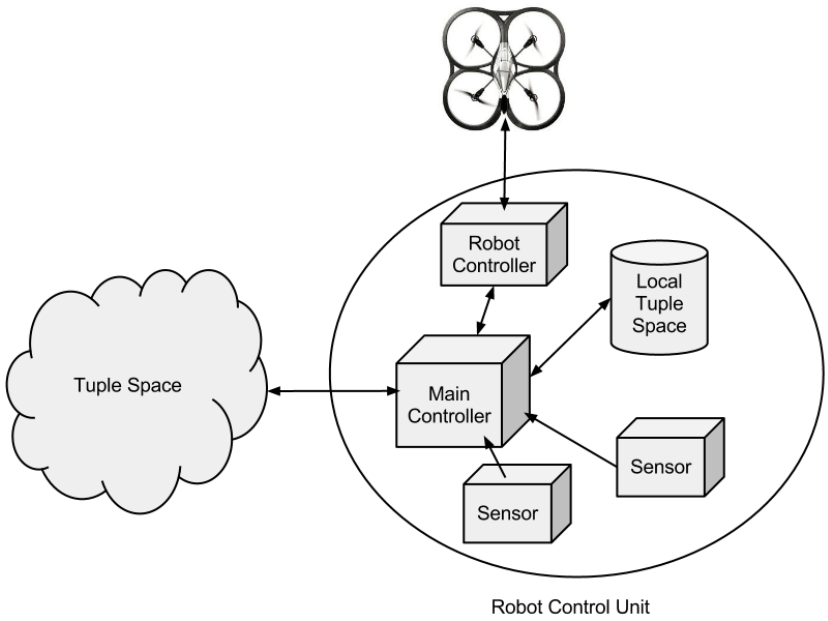

Figure 7.3: Design of robots.

UNIVERSAL CONTROLLER

The universal controller sets up a queue for pending tasks, so all tasks created specifically for this robot is retrieved and added to this queue. A thread is started to retrieve a task from the task queue and perform it. Only one task can be performed at any one time, except for certain special tasks.

The universal controller also has access to a list of tasks that can be run even though another task is in process. An example of such a task is starting to relay a data stream to a specific client, as relaying a data stream does not interfere with controlling the robot. It also has access to a list of prioritized tasks; tasks that should go before any current task, unless the current task is also a priority task. An example of this is an operator taking direct control over a robot. Note that both these lists need to be specified by the specific controller, as only it has the knowledge about tasks.

When a task is added to the task queue it is first checked if it can be run right away, and if so, it is performed right away. If not, it is checked for being a priority task. If it is, any task in process is halted and the new task is performed. If neither of these are the case, the task is appended to the task queue.

SPECIFIC CONTROLLER

The specific controller has the control logic for the specific robot, and must have access to an API to send control commands to the robot. In addition, it spawns a UDP server which receives commands sent directly from a client to the robot. The client may be any device in the system that is on the same network as the robot, and has access to the shared TS on that network. It is meant to be used by operators to send commands that needs so low latency that a shared TS cannot be used, such as controlling the robot. All logic concerning tasks are handled by the specific controller, so which tasks can be performed while others are in process and priority tasks, are specified here. Lastly, a thread is spawned which receives commands from the UDP server and relays them to the robot itself.

7.6 Compute Resource on a Cluster

The compute resource consist of a master node and many worker nodes. This section will first describe the design of the master and then the design of the workers.

7.6.1 Master

The master uses several threads and two separate servers. One of the threads are used to maintain workers. When the master receives a new task a thread is spawned to handle it. One of the servers is used to pass work to workers, while the second is used to pass results from workers and back to the master.

Workers sends heartbeats to the master over multicast, and if a worker isn't heard from for some time, it is assumed to be down. The master keeps the number of workers over a lower limit, and if the work-pool increases, new workers are spawned up to a maximum limit. If extra workers are spawned and then the work-pool gets drained, the master will kill off the excess workers, bringing the total amount of workers down to the lower limit.

The master receives tasks from a shared TS through a TSC. When receiving a new task, a new thread is spawned to handle that certain task. Each task must contain all necessary files to perform the desired task. These files need to contain all logic associated with the task, including retrieving jobs from the shared TS, splitting the job up and inserting into the work server, processing the job on worker nodes, retrieving and combining results, and inserting the final result into the shared TS.

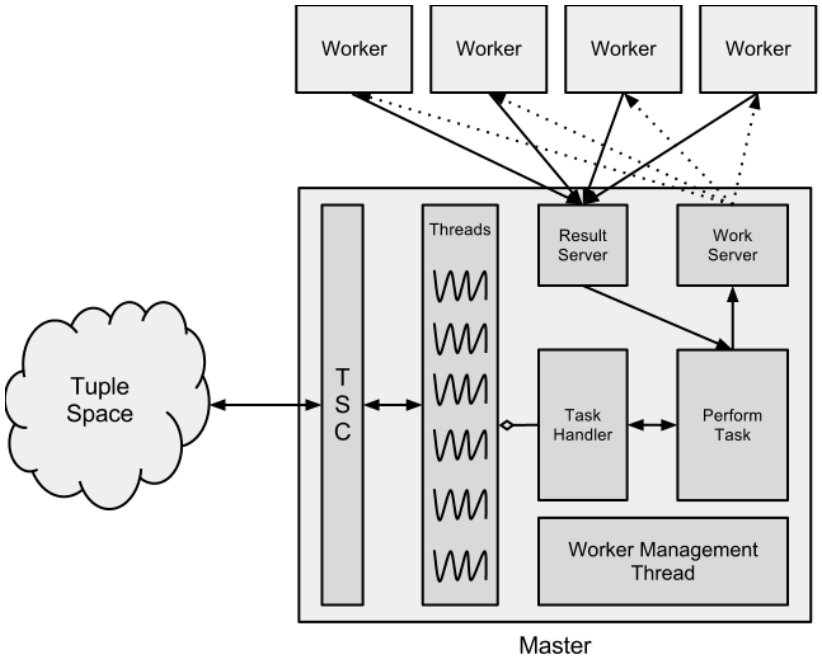

Figure 7.4: Design of the distributed compute resource. The dotted lines mean that the workers are the ones polling the work server for jobs.

WORK SERVER

The work server utilizes multithreading, so each request is handled by a separate thread. To handle any problems with concurrent access to the work-pool, the work-pool is protected by a lock. The work-pool works as a FIFO queue; the first job added is the first assigned to a worker. On a request, the thread acquires the lock before trying to retrieve a job from the work-pool. If the work-pool is empty, the thread releases the lock and blocks, otherwise it retrieves a job and returns it to the worker. When more jobs are added to the work-pool blocked threads are notified so they can retrieve a job.

The work server is also used to terminate workers. A special termination job is added to the server for every worker that needs to be terminated.

RESULT SERVER

The result server works mainly the same as the work server. It contains functionality to add results and retrieve results. The result-pool is protected by a lock in the same manner as the work-pool. The result-pool consist of several separate result-pools. one for each task. When a thread tries to retrieve a result, but none is available, the thread blocks.

7.6.2 WORKERS

The workers are rather simple by design. They retrieve jobs from the work server and performs them. All logic for performs the job on workers are specified in the files the master receives with the task. A worker should, by default, have no knowledge about how to perform any jobs at all. Details on how tasks and jobs are specified will be given in Section 8.6.

7.7 COMMUNICATING SEQUENTIAL PROCESSES

Communicating Sequential Processes (CSP)[24, 25] is a specification language used to describe interaction between several sequential processes. The processes run concurrently and communicate using channels. The processes may run on separate processors or they might run on a single processors with multithreading capabilities.

As the description of CSP is rather complex, description of the original concept behind CSP is omitted. For more information about this see [24, 25].

CHAPTER 8
IMPLEMENTATION

8.1 OVERVIEW

When deciding on which programming language to use there were two main considerations to take into account; the availability of a TS implementation, and the availability of an API for AR.Drone. In addition, it was preferred to use a high-level programming language, as it increases productivity. The TS that was initially selected was Linuxtuples (Section 4.4), which provides a Python API, but is implemented in C. However, due to problems discovered when running experiments (see Section 12.1.1), a new TS was implemented in Python. When an API in Python for AR.Drone was found, the choice fell on Python. The code is written in Python 2.6/2.7 in a Linux environment.

PyCSP is used in most cases where threads are spawned. Initially this was done because PyCSP was used more extensively. In the current state PyCSP is used less, but it provides an easy way to spawn new threads, and it makes it easier to ensure a graceful termination of processes. PyCSP is discussed later in this chapter.

All messages sent between devices use pickle unless specified otherwise. A message is pickled before it is sent and unpickled when received.

The pickle module is a standard Python module for object serialization and de-serialization. When pickling (or serializing) an object, the object hierarchy is converted into a byte-stream, while unpickling (or de-serializing) is the inverse operation. As pickle is Python-specific, any data that is pickled cannot be reconstructed in any other programming language. There are also certain restrictions to what data types can be pickled, see [26] for more information on this. The terms pickle and serialize will be used interchangeably in the rest of this thesis.

When protecting methods using a synchronization primitive, a decorator[1] is used. A decorator is a function that wraps other functions, being called before the

[1] Taken from SimpleTS at http://www.cs.uit.no/ johnm/code/teaching (March 27, 2012)

original function. The decorator then makes sure a specific lock is acquired before entering the methods it wraps, and releases the lock when leaving that method.

The rest of this chapter will look at the implementation of the different components in the system.

8.2 TUPLE SPACE HOST

As explained in Section 6.2 the TSH consist of two main components. First the TS server object is initiated. The TS server then tries to set up a local TS and, if successful, initiates an object which handles the creation of a dynamic NS (see Section 8.3 for more information on this), and the server start serving requests.

8.2.1 LOCAL TUPLE SPACE

The code to initialize and terminate the TS and, if necessary, specific access logic, differs depending on the TS implementation. For SimpleTS initialization is to simply create the object, and termination is to simply call the objects termination method. For Linuxtuples the subprocess module must be used to spawn a separate process running the TS on a predefined port. A connection is then set up on localhost to that port. Termination of the TS is done by using subprocess' termination command.

Any operations that are not supported by the default access logic must either be implemented there, or added to the TS specific access logic. For any operations to be available, they must also be implemented at the TSC.

The function to access a local TS simply takes a list of data values. The first value is expected to be a string specifying the operation to call, while the rest are assumed to be arguments to that operation. The string is then used to call the appropriate function. If the string does not match any function, an error is returned. The default access logic uses a decorator on all functions accessing the TS. The decorator makes sure the methods is wraps has to hold a lock before they can run.

SIMPLE TS

This implementation is inspired by SimpleTS[2], but most of the implementation has been taken directly from Lindypy[3]. As a result there is no architecture or design for this implementation. Due to the lack of any previous description on this

[2] http://www.cs.uit.no/ johnm/code/teaching (March 27, 2012)

[3] https://bitbucket.org/rfc1437/lindypy/src/2c0576344f5f/lindypy/TupleSpace.py (March 27, 2012)

implementaion, this section will describe the implementation with more details. As it uses the same API as Linuxtuples and the description is very tedious, this section can be skipped on the readers discretion. To ease the explanation, the important methods will be explained in full detail in pseudocode.

The TS uses two dictionaries, *tspace* and *lenidx*, and a list, *colidx*, to store tuples. The dictionary *tspace* uses a counter, which is incremented for each tuple added to the TS, as index, and the element is the actual tuple. *lenidx* takes the length of tuples as index, and the elements are sets of tuple indices. So the element at index ”6” is a set containing all indices for tuples with 6 elements.

The list *colidx* is a list of dictionaries, where the dictionaries contains sets. It is used to map tuple elements to tuple indices. A dictionary at index i maps all elements found at index i in a tuple to the indices of tuples containing this tuple.

Since the implementation is based on Linuxtuples, None is used as a wildcard. This means that tuples should not contain any elements that are None.

Put

Assume we want to insert a tuple t with n elements $e_0, ..., e_n$. First the counter is incremented and used as the index for t. It is inserted into the set found at index n in *lenidx*. When mapping t with *colidx* we need n dictionaries in the list. For all $j \in [0, n]$ we use e_j as index in the dictionary found at index j, and the element in the dictionary is the hash index of t.

```
def put(tuple):
  counter++
  tspace[counter] = tuple

  if length_of(tuple) not in lenidx:
    lenidx[length_of(tuple)] = set()
  lenidx[length_of(tuple)].add(counter)

  while length_of(colidx) < length_of(tuple):
    colidx.append( dictionary() )
  for index, value in tuple:
    if value not in colidx[index]:
      colidx[index][value] = set()
    colidx[index][value].add(counter)
```

Listing 8.1: Inserting a tuple into the tuple space. Counter is used as the index of the tuple in tspace, which is a dictionary that stores the actual tuple. Index and value, used in the for-loop, are the index in the tuple and the element in the tuple corresponding to that index.

Retrieving Tuples

Retrieving tuples from the TS is based on one method which takes a template and a number as input. The template specifies how the tuple should be, and the number specifies how many matching tuples should be returned. Assume we want to retrieve a tuple matching the template t with n elements $e_0, ..., e_n$. First the set, s, at index n in *lenidx* is retrieved, giving all tuple indices for tuples with right length. Each element in the t is then checked. If it is a wildcard the element is ignored, as it matches anything. All elements that are tuples that contain a wildcard or a formal parameter are stored in lists to be checked later.

All $j \in [0, n]$ the elements e_j in t that are not wildcards, tuples containing wildcards, or formal parameters, are checked against the dictionary at their respective indices j in *colidx*. If any e_j is not an index in the dictionary, there is no matching tuple in the TS. If e_j is an index in the dictionary, the intersection is taken between the set at that index and the set s, replacing s with the resulting set. After the intersection the set s will contain all tuple indices for tuples that match the template so far. If the the set s ever ends up empty, there are no tuples matching the template.

After all elements have been checked against *colidx*, the set s holds indices to all possible matches. The indices in s are iterated and their respective tuple found, and each element that corresponds to a field in the template that is a formal or a tuple with wildcard are checked. If the template element was a formal, the element in the tuple is checked to see if it is of that type. If the element was again a tuple with a wildcard, each element is checked to see if the template element matches the tuple element. If all elements match, the tuple along with its index is appended to a list. Once enough matching tuples have been found, or the set is empty, the list containing matching tuples is returned.

When reading or getting a tuple, a template is passed along with the integer 1, specifying that only a single tuple is needed. If a tuple is returned, and it is a get, the tuples index is removed from all the sets and the tuple is removed from the TS dictionary. If it is only a read, this is not done. Count and dump iterates all the input templates and finds tuples matching each template, aggregating results between. When matching tuples to a template, all matching tuples are retrieved. If no templates are given, all tuples in the TS are returned.

```
def retrieve(template, n):
  if length_of(template) not in lenidx:
    return []
  candidates = lenidx[length_of(template)]

  wildcard_tuples = []
  formal_parameters = []

  for index, value in template:
```

```
    if value == None:
      continue
    else if value is formal parameter:
      formal_parameters.append( (index, value) )
    else if value is tuple and value contains wildcard:
      wildcard_tuples.append( (index, value) )
    else:
      if value not in colidx[idx]:
        return []
      else:
        candidates = candidates.intersection( colidx[index][
            value] )
        if candidates is empty:
          return []

  if candidates is empty:
    return []

  result = []
  foreach tuple_index in candidates:
    tuple = tspace[tuple_index]
    if tuple matches elements in wildcard_tuples and
       formal_parameters:
      result.append(tuple)
    if n > 1:
      n--
    else:
      return result
```

Listing 8.2: Retrieving tuples from the tuple space. Candidates is a set of tuple indices. Index and value are the index and corresponding element in the template. Elements whose value is a tuple with a wildcard in it are stored in wildcard_tuples, while elements whose value is a formal parameter is stored in formal_parameters. These are matched at the end when all candidate tuples have been retrieved.

8.2.2 Tuple Space Server

The TS server uses TCP/IP sockets. File sockets are used both for receiving data and sending results back. First the size of the message is sent over the TCP socket, and then the message is sent over the file socket.

The server is implemented using the SocketServer module, which is a standard module in Python 2.6 and 2.7. It utilizes the multithreading support from the module to support concurrent access to the local TS. Handling any problems with concurrent access is left the access logic of the TS, or the TS implementation itself. As explained in Section 8.2.1, the default access logic handles concurrent

access by default.

To allow remote TSCs to locate the TSH, the server spawns a thread that sends a heartbeat to a NS. It is assumed that the heartbeat is propagated from this NS to all other NSs it knows about. When successfully sending a message, a list of all known NSs is returned. This list is stored and only changes when a new list is returned.

When a request is received, the connection is kept open until either the client closes the connection, or some error occurs and the server closes the connection.

8.3 Name Server

The NSs use TCP/IP and are built over the SocketServer module in Python. The IP address of TSHs and NSs are stored in separate dictionaries, where their IP address is the index and a timestamp is the entry. When sending a heartbeat using multicast, messages are not pickled as only short strings are sent.

When a connection is established with a NS, it assumes that it will receive a request right away. The requests are assumed to be a list whose first element is a string referring to the requested operation, and the remaining elements are arguments to this operation.

Whenever a TSH is added to a NS, the message is relayed to all other NSs. Since heartbeats simply are (re-)adding a TSH to a NS, heartbeats are relayed this way too. If a NS receives a request to add a NS that it has no current knowledge of, it sends all TSHs to the new NS.

8.3.1 Static Name Server

The static NSs don't use multicast at all, as it assumed that there are not other servers on its subnetwork. As the name suggests, it is also assumed that static NSs doesn't have a varying IP address, so they should be available over TCP/IP. Note that the IP address of static NSs must be provided to TSHs and static NSs initially, or else they will not be found.

Static NSs come with a simple shell with some commands; the same interface provided via TCP/IP can be used by the shell, along with a command to gracefully terminate the server.

8.3.2 Dynamic Name Server

Each TSH will start a thread that monitors NSs. This thread will listen to the multicast heartbeat from dynamic NSs and will ping static NSs. If there are too few NSs detected, the thread spawns a new NS.

Dynamic NSs can also receive requests over multicast. If a TSH or TSC do not know about any NSs, it sends a multicast message with a specific port number. When a dynamic NS receives a message with a port number, it sends its own IP address to the senders address on the specified port number. This way devices can locate NSs, and thus find TSHs, on the subnetwork without any prior knowledge of any other devices.

If a dynamic NS detects that there are too many NSs, it will use a simple determinant algorithm to decide if it should terminate.

8.4 Tuple Space Client

The TSC is comprised of two components, the TSC itself and the access logic. The TSC holds methods building the API for the shared TS and functions for inserting tuples into the TS. The TSC also spawns a thread for handling insertion of tuples into the shared TS. The access logic holds functions used to send requests to the shared TS.

8.4.1 TSC

The methods building the API are put, get, get_blocking, read, read_blocking, dump, and count. The blocking versions of get and read simple call their respective non-blocking versions. If no matching tuple or error is returned, the method waits for a short duration before calling its non-blocking versions again. This loops until a result or error is returned.

Put, get, and read all take a single argument that have to be a tuple. Dump and count both take a list that should either be empty or contain only tuples. If the argument to these functions are not correct, an error is returned.

Put is protected by a lock that must be acquired before running the function. A decorator is used to handle the synchronization. If the argument to put is valid, it is appended to a FIFO queue, represented as a list, and one thread waiting for the lock is notified to wake it up (there is only one thread that may block). The thread that handles insertions will retrieve tuples from the FIFO queue and try to insert it into the shared TS. For a single tuple, a loop is run until it is successfully inserted. If the tuple fails to be inserted, the delay between each attempt increases gradually up to a maximum delay.

8.4.2 Access Logic

The access logic is comprised of two main methods, call and call_aggregate. The first is for operations that only need a successful response from a single TSH, the

latter is for operations that need to aggregate results from all TSHs. Call receives a single argument, which is the data that is to be sent to TSHs. Once a successful result is returned from a TSH, this result is returned to the caller.

If a connection is already open when contacting a TSH, it is used first. If there is none, or that TSH cannot fulfill the request, all TSHs are found. When retrieving TSHs, a NS is contacted. When contacting a NS, a list of TSHs and a list of NSs are returned. These are stored and only updated the next time either of these are received from a NS. If a list of TSHs was retrieved a short time before, it is used instead of retrieving a new list from a NS.

When setting up a new connection to a TSH any existing connection is closed. A TCP/IP socket, as well as two file objects, one for reading and one for writing, are created. The connection is not closed when a transmission is complete; it is left open until a new connection is set up, or until it has been unused for some time. A separate thread is used to monitor the use of the socket, and close it if it is not used within a certain time (currently 10 seconds).

The aggregate method works similar to the call method. It takes the data that is to be sent, a result data type, and an operator as arguments. The result is an empty instance of the the data type that is expected from the TSHs, and the operator is the function used to aggregate results from different TSHs. For count the result argument is 0, while the operator is addition, while for dump the result argument is an empty list and the operator is addition.

8.5 Robot Control Unit

The robot used for testing is the Parrot AR.Drone (version 1), developed by Parrot[18]. For specifications of the AR.Drone v1.0, see Appendix A. All code have been written for, and tested with, the AR.Drone. Due to problems with implementing autonomous tasks for the drone, along with lack of time, parts of the implementation is not complete, and much of the code should be expected to have bugs. This will be discussed further in Chapter 12.

8.5.1 Main Controller

The main controller is implemented as a Python class. It has an instance of a TSC and an instance of the robot controller. When a task is received from the shared TS and validated, it is appended to the robot controllers list of tasks. The main controller also holds a dictionary that maps a task ID to the sender of that task.

8.5.2 Universal Controller

The universal controller is implemented as a Python class. Any specific robot controller *must* inherit from this class.

The universal controller holds a list of tasks that can be performed at any time, a list of tasks that should take priority, and a list of tasks pending to be performed. Tasks have to contain two elements: task specification and task ID. The task specification must be represented as a list, and the first element must specify the function to run, represented as a string. Any remaining elements in the list is passed as arguments to the function.

A CSP channel is set up to send messages from the robot controller to the thread executing a task.

8.5.3 Open Source Computer Vision

Lets first define two terms which are related, but have distinct purposes: image analysis and image processing.

- Image analysis is the extraction of meaningful information from images, mainly utilizing image processing techniques.
- Image processing is a process where different signal processing techniques are applied to an input image, and the result may either be an image or a set of properties related to the input image.

Before describing the specific controller, we'll first look at the functions used for image processing. All analysis of images are done by using a open source image processing library called Open Source Computer Vision, or OpenCV[27]. OpenCV provides a variety of powerful tools for processing and analyzing images. The following paragraphs will describe the implementation of functions using openCV to analyze the video stream from the drone.

IMAGE PROCESSING

First the functions using openCV to process and analyze images will be described. The following will go into detail on how the image analysis is done.

Utility Functions

Some utility functions are provided to perform common operations. A simple function is used to open an image from a RGB-string, and a simple function is added to show an image passed as an argument. These are mainly for use in scripts that does not include the openCV-module.

Colour	Hue Value
Red	$(172, 179] \cup [0, 11]$
Orange	$(11, 25]$
Yellow	$(25, 35]$
Green	$(35, 83]$
Teal	$(83, 92]$
Blue	$(92, 131]$
Purple	$(131, 148]$
Pink	$(148, 172]$

Table 8.1: Hue range for specific colours

A function to find a white object and return it as a binary image is provided. This function converts the image to HSV (Hue Saturation Value) and uses threshold operations to find the white areas in the time.

Another function determines if a contour from a list of contours touches the any of the edges of the image. It takes the image and the list of contours as argument. Then it simply loops through all points in the contours, checking if they are within a certain distance from the edges of the image. If any point is too close to an edge, the list of contours is said to be touching the edge.

Two different functions to filter out contours are provided; one to remove all contours smaller than a specified size, the other to remove all but the largest contour.

Find Object

This function tries to detect objects with specific colour. The function is based on the openCV tutorials found at [28]. First the image is converted to HSV, before the different colour bands are split into separate images. A string specifying which colour to look for is converted to a value range which represent that colour in the hue band. When representing a colour in the HSV colour space, it is common to represent each of the bands as a value in the range 0-255. However, in openCV is represented by a integer in the range 0-179, which complicates the process of finding the hue range which represents a certain colour. A tool found at [28] was used to loosely find the hue-ranges for different colours, which can be seen in Table 8.1.

The hue image is then filtered on the range to produce a binary image. A thresholding operation is used to set each pixel which is outside the colours range to 0, while all pixels within the range is set to 255 (producing a black image with white shapes where the hue value is between the desired range). The saturation and value of the image is also filtered using the threshold operation, to remove

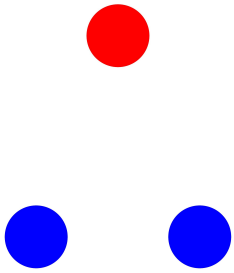

Figure 8.1: Marker for finding a direction.

very bright and very dark colours. Filtering the saturation and value images provide two more binary images. These images are combined into a final binary image.

Follow Object

This is based on a script found at [29], with only small modifications, mostly removing code that is excessive when this is used in combination with the function to find an object with a certain colour.

This function uses the contour of an object which it is to follow. If there is no contour in the image, the movement the drone should take is set to 0. If a contour is present, it is used to find the center of the contour mass. The distance from the center of the mass to the center of the image is found, before a PID controller (Proportional-Integral-Derivative)[4] is used to calculate the movement the drone should take to center the mass in the image.

Note that the movement calculated only takes roll and altitude (x and y) into consideration, so yaw is set equal to roll. The pitch is calculated from the size of the contour.

Detect Marker

This function tries to detect a marker on the floor, using the camera facing downwards. A marker must consist of two blue dots and a red dot, forming an isosceles triangle, on a white sheet of paper. Figure 8.1 shows how the marker should look like.

The first steps to finding the marker is to find any white areas in the image.

[4]PID Controller from http://www.control.com/thread/1026159301 (March 6, 2012)

Any white shapes found are filled, so there are no holes in the shapes. Any red and blue shapes are then found, giving two binary images, one for each of the two colours. The white binary image is then combined with these two binary images, giving one binary image for blue shapes on a white background, and one for red shapes on a white background. The angle that the marker points in is found using basic trigonometry. The code to find the angle formed by three points is based on an openCV example[5]. The function returns the angle and the center of the white mass. If the center of mass on either the red or the blue shapes cannot be found, the angle is set to False, and returned along with the center of the white mass.

IMAGE ANALYSIS

The following will describe functions used to analyze the video stream to extract any useful data. The functions here utilize the functions described in Section 8.5.3.

Find Object

This function detects objects of a given colour. It utilizes the image processing functions to detect any objects with the specified colour, giving a binary image. Small shapes are filtered out, and then the largest contour in the image is found

Follow Object

This function is very alike the previously described function. The analysis is the same as when finding an object, although after finding the largest contour, this contour is passed on to the image processing function to find movement variables. The movement variables are returned to the calling process.

Pop Balloon

This function works similar as the function to follow an object. The only difference here is that the area of the object is found and returned with the movement variables.

Detect Marker

First the marker is detected, giving the center of mass of the marker and its angle. If specified to, a function is called to find movement commands to center over the marker, else the movement commands are set to 0. The angle, the center of the marker, and the movement commands are returned.

[5] https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/python/squares.py?rev=2785 (March 6, 2012)

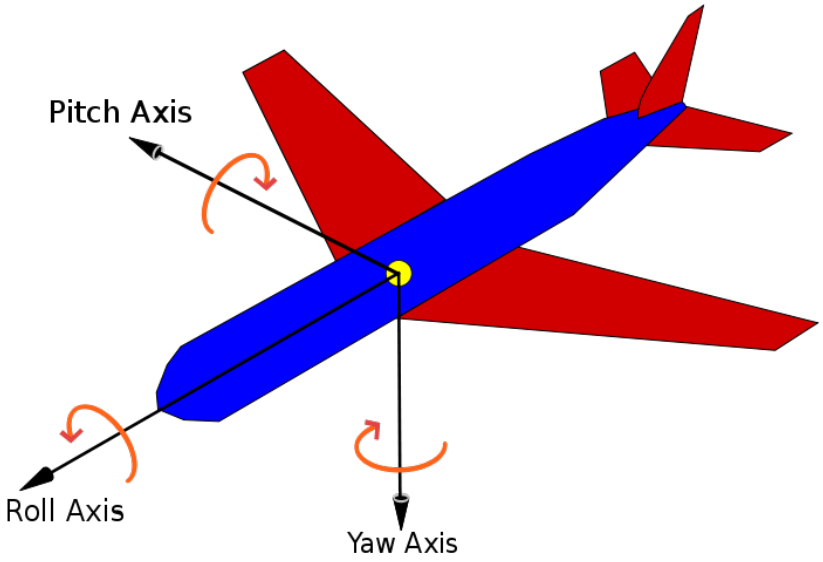

Figure 8.2: Illustration of the position of the axes that an aircraft can rotate around. The illustration was taken from wikipedia[6].

8.5.4 Specific Controller

The specific robot controller holds an instance of the AR.Drone controller class, or the drones API, used to send commands to the drone. A simple UDP server is set up to receive commands. Any commands received are passed to a thread through a PyCSP channel. This thread then sends them to the drone via its API.

Upon initialization, the list with tasks that can be performed at any time, and the list with priority tasks, is set.

AR.DRONE API

The API used to control the AR.Drone is based on the API provided by Bastian Venthur[30]. The file handling video decoding and the file handling the network traffic remains mainly unaltered, so only the additions to the main API is mentioned.

The most important functionality is the control function. It takes a list as parameter, which is assumed to be a list or tuple containing two elements: control message and control state. The control message consist of four floating point values, which are roll, pitch, yaw and altitude. The three first control the movement

[6] Wikipedia article on Aircraft principal axes, http://en.wikipedia.org/wiki/Aircraft_principal_axes (March 6, 2012)

around the three principal axes, as seen in Figure 8.2. The last control the altitude of the drone. All these values specifies the velocity of the movement the drone gets, from 0% to 100% of a set maximum velocity on the different four axes. If one of the values are set to 1.0 the drone will move with the maximum velocity to one direction on that axis, while -1.0 will make it move with maximum velocity in the other direction on that axis. As an example, a positive altitude will make the drone gain altitude, while a negative value will make the drone lose altitude. For more information on these four values, please refer to the AR.Drone SDK Developer Guide, chapter 6.6, which can be found at [19].

The control state is a dictionary which holds the state of the drone, currently only if the drone should be flying and if there is a user emergency. A user emergency is when the drone has received a command from outside to enter emergency mode, which will cut all engines immediately. If the drone is in an emergency state, but there is no user emergency, then the emergency will be resolved. If the drone is not in emergency mode, but the control state received has the emergency flag set, the drone will enter emergency mode and cut all engines.

If there is no emergency at all, the start flag is evaluated. If the drone is not flying, but it is set to take off, a take off command is sent. If the opposite is true, a land command is sent. Lastly, if none of these cases are true, the movement commands are sent.

Functions to set the maximum height the drone is allowed to fly at, to change the camera mode, and functions to return altitude, direction, and velocity are also added.

BASIC TASKS

There are many fairly basic tasks implemented, which will be explained briefly here.

Stop Task

This tasks is simply implemented to stop any on-going autonomous task. It uses the existing CSP channel to send a certain message to the on-going task. This task is allowed to be performed at any time.

Start Video Stream

This task takes start relaying the video stream to a certain device. It takes the IP address of the device as an argument and adds this to the drones list of devices that the video stream should be relayed to. This task is allowed to be performed at any time.

Stop Video Stream

This task stops relaying the video stream to a certain device. It takes the IP address of the device as an argument and removes it from the list that the drone relays the video stream to. If the device is not present in the list, nothing is done. This task is allowed to be performed at any time.

Switch Camera Mode

This task switches the camera mode of the video stream from the drone. The function takes an IP address and the camera mode as arguments. If there is no controller of the drone, or if the requester IP address matches the controller IP address, the request is allowed to change camera mode. This task is allowed to be performed at any time.

Start Controlling Drone

This task lets a certain operator take direct control over the drone. The function takes the IP address of the device requesting direct control as an argument. If nobody is registered as controlling the drone already, the requester is given control. If an IP address is registered as having direct control, the request is refused, unless the requester IP address is the same as the IP address already controlling. When control is given to the requester, the UDP server of the drone registers the IP address address of the controller, and the drone start relaying the video stream to the controller. This task is regarded as a priority task.

Stop Controlling Drone

This task stops the controlling of the drone. It takes an IP address as an argument and checks it against the current controller of the drone. If they are not the same, the request is refused. If they are the same, the IP address registered at the UDP server is removed. This task is regarded as a priority task.

AUTONOMOUS TASKS

There are three autonomous tasks implemented for the drone. As the only available sensor that can be used for any autonomous tasks is the cameras, all autonomous tasks are based on analysis of the video stream. All autonomous tasks are implemented by utilizing the image analysis functions explained in Section 8.5.3. All tasks take a CSP channel reader object as input, to receive any necessary messages. They can all be terminated through a certain message being received via the CSP channel. It is assumed that the drone is landed when any of the autonomous tasks are started, and it is assumed that the drone should land afterwards.

Follow Object

This task takes the requester IP address and a colour as arguments. It is important to note that it only takes colour and size into consideration; it will follow the largest object matching the given colour, but the shape must cover a certain percentage of the screen.

First is searches for an object matching the given colour. It simply takes off and then turn around its yaw-axis (see Figure 8.2), searching for a fitting object. The function used to search for an object returns either when it receives a stop-command via the CSP channel, when it finds an object, or when it has turned approximately 360 degrees.

If any matching object was found another function is run, which uses the image analysis function to follow an object. If the movement commands returned from this function are all 0, it is assumed that no object was found. If this stays true for a certain amount of time, the function returns and the task is finished.

Pop Balloon

This task works basically the same way as the above task, so the differences will be explained. The purpose of this task is to detect an object with a specific colour and, assuming that object is a balloon, pop it.

When searching for an object (hereafter assumed to be, and referred to as, a balloon), the drone searches for a balloon at several altitude levels. The drone is set to search 360 degrees around its yaw-axis every 100 mm, from 200 mm to 1800 mm height. When a balloon is found, another function is run to make the drone pop the balloon, using the image analysis function describe in Section 8.5.3.

Between each iteration, the balloon area in the image is stored. When the result from the image analysis is returned, the old balloon area is compared to the new balloon area. If there is a big difference between them, the drone tries to back up a bit to try to detect the balloon again. If the balloon cannot be found, it is assumed that it was popped, and thus it will consider the task completed. If the balloon can be found, the movement commands are sent to the drone.

Follow Waypoints

This section describes a unfinished task. The task takes a virtual map and a set of waypoints as arguments, and makes the drone fly through the waypoints. It is assumed that a set of markers are placed in known positions and that all markers are pointing in the same direction. The drone must initially be placed over a marker, and know which marker it is placed at. The detect marker method describe in Section 8.5.3 is used to detect markers on the ground, and this data, along with data from the drones IGS, is used to build up the virtual map.

The data from the IGS is used to estimate changes in the position of the drone,

while the markers are used to correct any errors in the position estimation and direction of the drone. Due to problems which will be discussed in Chapter 12, this task does not work, and the implementation was therefore not completed.

8.6 Compute Resource on a Cluster

The compute resource was implemented solely for experimental reasons, so the implementation is not robust. Many solutions used in the implementation is chosen simply because it was easy and straightforward.

8.6.1 Master

Both the work and result servers are implemented over the SocketServer module. When spawning extra workers, ssh is used to execute the worker script on a node. The node is selected from a list using round robin. The master utilizes the fact that the file system (Network File System, or NFS) on the cluster distributes files over all nodes.

When the master retrieves a new task from the shared TS, it uses PyCSP to spawn a new task handler thread. The task handler retrieves the needed files from the task and runs the code to perform the task.

One of the files have to contain a class named ”task” which should take the following arguments, in their respective orders: result server, work server, task sender, task ID, task specifications. This class should then contain a method named ”main”, which is the method that is run. When this method returns, the task is assumed to be finished. The task specification can be whatever needed to perform the task as needed, as long as it can be inserted into the TS. The task consists of one or many jobs. The jobs are what is actually passed to the work server and then to workers.

WORK SERVER

To handle concurrent access to the work-pool, all methods that do any operations on it is protected by a lock. A decorator[7] is used, so all methods encompassed by the decorator have to acquire the lock before they can run their own code. The decorator is implemented using the threading module.

When a request is received, the server tries to retrieve a job from the work-pool. The work-pool is simple a list, which serves as a FIFO queue; jobs are appended to the list, and when jobs are retrieved they are popped from the front of the list. If it is empty, the thread blocks, leaving the connection to the worker

[7] Taken from SimpleTS at http://www.cs.uit.no/ johnm/code/teaching (March 27, 2012)

open. The lock provides a method that releases the lock and blocks the thread until it is notified. If jobs are added to the work-pool, the server notifies blocked threads. If a single job was added to the work-pool, only one thread is notified, but if more than one job was added, all blocked threads are notified.

If a worker receives a job from the work server that is None, it is a notification that the worker should terminate. This can happen in two cases; either by the master killing off excess workers or by the master terminating.

RESULT SERVER

The result server contains a result-pool represented by a dictionary, where the index is the task ID and the entry is a list of results. So the result-pool is made up of many separate result-pools, one for each task. The result-pool is protected the same way as the work-pool, by a decorator which makes sure a lock is acquired before accessing the result-pool. When retrieving a result, the first element in the tasks result queue is returned. So even if many results are available, only one is returned at a time. If no result is available, the thread blocks. Whenever a new result is added, all blocked threads are notified. A result is the output data from performing a job.

The result server only support one result being added at a time.

8.6.2 WORKERS

The workers retrieves jobs from the work server. Prior to receiving jobs, the worker knows nothing about it. When a job is received, the module specified in the job is imported, unless it has already been imported. This module must contain a method named ”perform_job”, which is responsible for performing the job. This method should take two arguments: job_info and job_data. The first of these should contain information about the job, such as task ID. The latter should contain any data needed to perform the job. This method should return the result, which is sent to the result server. The result should be a list or tuple where the first element is the task ID and the second element is the result itself. Task ID is the ID of the task that the job is a subset of.

8.7 PYCSP

PyCSP[31, 32] is an implementation of CSP in Python. Just as in conventional CSP, PyCSP programs commonly consist of many CSP processes that communicate over channels. This section will give a brief description of PyCSP. PyCSP

v0.7.1 was used[8].

8.7.1 Channels and Channel Poisoning

In earlier versions of PyCSP many different channel types were supported. However, after feedback that the Any2Any-channel was basically the only one being used, the interface was changed to only support this one channel type.

To retrieve a channel end two functions are provided: channel.reader() and channel.writer(). When a process has retrieved one end, it can use this end to communicate with other processes using the same channel.

Two methods for closing a channel are provided: poison and retire. Poison is called on a channel, and any following read or writes to that channel will throw a poison exception, which can be caught and handled. Retire can be used to leave a channels end (channel ends are the reading end and writing end). When all parties on one end have left, so there is either no reader or no writers left, the other end is retired too.

8.7.2 Alternation and Guards

Alternation supports two calls: execute and select. Execute waits for a guard to complete before executing the choice function associated with that guard. Select chooses a guard and returns a tuple of the guard and, if the guard was an input-guard, the message that was read. PyCSP support four types of guards; input, output, timeout, and skip. Timeout guards lets alternation to complete after a set timeout. Skip is always set to true, which will cause an alternation to always be able to return immediately.

8.7.3 Processes

For a process to be run as a CSP process in PyCSP, it have to be encapsulated in a process decorator, a Process class. A Python function is passed to a instance of the Process class. The function then implements the CSP process.

Three constructs are provided for running PyCSP processes: Parallel, Sequence and Spawn. Parallel takes a list of instances of a PyCSP process and executes them. The processes runs concurrently, but the executing process waits for all the PyCSP processes to be terminated before continuing. Sequence works the same way as Parallel, but the processes are executed sequentially, so only one runs at a time. Spawn works just as Parallel, but returns to the executing process after starting the CSP processes.

[8] http://code.google.com/p/pycsp/, May 11, 2012

8.7.4 Use of PyCSP

Initially PyCSP was considered to have a bigger role in the implementation, which is the reason it is used. However, it is now mainly used to ease the creation of new threads by using the spawn command. One reason for using PyCSP to spawn threads is that only a minor change is needed to make to spawn new processes instead. This can enable better performance, as it removes the concurrency problem introduced by the Global Interpreter Lock (GIL). In the RCU channels are also utilized along with alternation and guards. This was done because it makes it easier to pass messages from the robot controller and ongoing autonomous tasks.

CHAPTER 9
METHODOLOGY

The research presented in this thesis follows a systems approach. This approach consist of five stages (see Figure 9.1): idea, architecture, design, implementation, and experiments. The first stage, the idea, describes the goal of the research and its intention at an overall level. A system is created to evaluate the idea and to observe its implications on real hardware. Creating the system consist of the three stages: architecture, design, and implementation. Architecture describe the interaction between the main components in the system. Depending on the architecture, a design is chosen. The design describes the components in the architecture, defining how they should be realized. The implementation realizes the system on hardware.

In the last step, experiments, performance measurements are defined and carried out in a controlled manner, demonstrating the system on real hardware and giving data to evaluate the system. The results of the experiments are analyzed and used to draw conclusions. Depending on the conclusions, the idea, architecture, design, or implementation is revised.

There are many different architectures that can realize the idea, just as there are many designs that can realize the architecture, and many implementations that can realize the design. This means that if experiments don't give the expected results, it can be the result of the final path down to the implementation, which is based on many different decisions. It is still worthwhile to report the idea, results, and the chosen path, because it demonstrates the impact of a specific path, and might give useful insight for other research on the same area.

9.0.5 METRICS

In this thesis the following metrics are used for performance measurements: CPU-load, memory usage, network bandwidth, and latency.

When measuring latency in the system the time module is used. The time is stored at two different points in the code, preferably immediately before and after

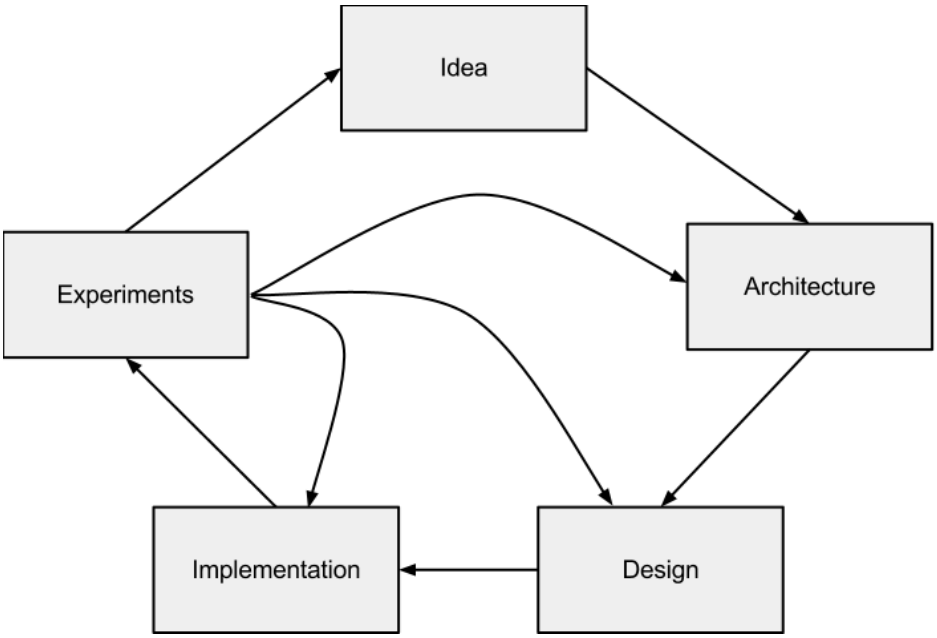

Figure 9.1: Systems research methodology.

the code snippet that should be measured (see Listing 9.1), and the difference between these are logged by writing to a file. It is important to note that one should be careful when using the time.time method for timing, as its output might vary between systems; for some systems it might not provide better precision than 1 second. However, for UNIX systems, time.time() provides time with high precision.

```
t = time.time()
(...) Code snippet (...)
Total_Time = time.time() - t
```

Listing 9.1: Measuring latency for code snippets.

When measuring the network latency the roundtrip is measured: the time for sending an empty string from a client to a server, and then back again, is measured. When calculating the standard deviation and mean value, numpy was used.

9.0.6 CPU Load

CPU load describes the time-period a process has been running on the CPU divided by the total elapsed time of the measurement, multiplied by 100, which gives it in percentage. This means that if a process has been allocated for 500 milliseconds over a total of 1 second, the CPU load is 50 percent. Multicore processors can have a CPU utilization of 100 percent multiplied by the number of

cores, and processors with HyperThreading enabled can reach 200 percent for a single core. CPU load can be split into user-level load and kernel-level load, where user-level is time spent in user space performing non-privileged instructions, and kernel-level is time spent inside the kernel on behalf of the process. The interesting aspect with CPU load is just to get an indicator on how much load the system puts on the CPU. The Unix command ps (process status) is used to find the CPU load (see Listing 9.2). This number is somewhat inaccurate, as it sometimes can report over 100 percent CPU load, but it is accurate enough to give the desired indication.

```
ps -p PID -o %cpu -o %mem -o rss -o maj_flt -o vsize
```

Listing 9.2: Measuring resource usage.

9.0.7 Memory Usage

Memory usage is the total amount of memory a process has allocated. As with CPU load, only an indicator is desired, so the Unix ps command is used to retrieve memory usage (see Listing 9.2). The memory usage is reported in percentage, which means it is the memory allocated by the process divided by the total amount of addressable memory available , multiplied by 100 percent.

In addition to this, the resident set size (RSS), virtual memory size, and major page fault count is logged. The RSS is the non-swapped physical memory, in KB, used by the process. The virtual memory size is the virtual memory, in KB, used by the process. A major page fault is when the page resulting in a page fault is not loaded in memory, resulting in the page being retrieved from disk.

9.0.8 Network Bandwidth Usage

Network bandwidth usage is the number of bytes sent and received by a process within a certain time period, given in bits per second. This only covers the data sent to the socket, not additional data added by underlying layers. In this thesis network bandwidth is measured and logged inside the process itself, by logging the amount of data sent along with the latency of the send and receive operations.

9.0.9 Latency

Latency is the metric describing the delay between two events in the system. In network communication, latency is the time to send a message between two parties (send and receiver). Latency of an operation is the time from the operation is called until it returns. Latency is measure in seconds/milliseconds.

CHAPTER 10
EXPERIMENTS

10.1 OVERVIEW

The impact of tuple size, TS population, and number of participants (both TSHs and TSCs) are the focus of the experiments. To test this a series of 8 experiments were run. One experiment was run to test the performance of the name server, one experiment was run to determine the bandwidth and evaluate the streaming server, and two experiments were run to evaluate the local tuple space implementation (SimpleTS). The last four experiments were to evaluate the TSC and TSH implementation.

Table 10.1 and Table 10.2 show the specifications for the computers used in experiments. Which of the two setups are used will be specified for each experiment. The network is a cabled gigabit network.

When discussing get and read, which takes a template as argument, tuple and template will be used interchangeably to refer to the input argument. When discussing count and dump, which take a list of template as arguments, tuple, template and templates will be used interchangeably. This is because, unless specified, the argument to get and read is the same tuple as the one inserted by put, and the list of templates to count and dump only holds one template, which is the same tuple.

10.2 STREAM SERVER

When evaluating the stream server a single server and a single client was used. The client generated a set of strings with increasing size, from 128 KB to 256 MB, doubling the size for each new string. Each string was sent 1 000 times to the server and then sent back to the client, creating a new connection every time. The total latency from setting up the connection until closing the connection, as well as the roundtrip latency was logged on the client, while the time from a connection

Computer	DELL Precision WorkStation T3500
Operating System	CentOS 5.7, 32-bit
CPU	Intel Xeon E5520, 2.26 GHz
Level 1 cache size	4 x 32 KB instruction caches 4 x 32 KB data caches
Level 2 cache size	4 x 256 KB
Level 3 cache size	8 MB
RAM	12 GB, 1066 MHz DDR3 ECC
HDD	500 GB 7200 rpm S-ATA
GPU	QUADRO FX-580 512 MB

Table 10.1: Experimental rig 1

Computer	DELL Precision WorkStation T3500
Operating System	CentOS 5.7, 32-bit
CPU	Intel Xeon E5520, 2.53 GHz
Level 1 cache size	4 x 32 KB instruction caches 4 x 32 KB data caches
Level 2 cache size	4 x 256 KB
Level 3 cache size	8 MB
RAM	12 GB, 1066 MHz DDR3 ECC
HDD	500 GB 7200 rpm S-ATA
GPU	QUADRO FX-580 512 MB

Table 10.2: Experimental rig 2

was accepted until it was taken down was logged on the server. Listing 10.2 shows how the latency was logged on the client; the time from before the socket is created to the socket is closed is the total transmission latency, while the roundtrip latency is from before the data is sent until the data is received again.

```
While i < 1000:
    Time_Start = current_time()
    Set up connection (...)
    Bandwidth_Time_Start = current_time()
    Send data (...)
    Receive data (...)
    Bandwidth_Time = current_time() - Bandwidth_Time_Start
    Close connection (...)
    Total_Time = current_time() - Time_Start
```

Listing 10.1: Measuring latency on client for stream server.

On the server the total transmission latency is measured by taking the time before the thread handling the connection is created, and just before the thread is about to terminate. Listing 10.1 shows how the latency is measured on the server.

```
Accept connection (...)
t = current_time()
Spawn thread  (...)
Set up sockets  (...)
Receive data  (...)
Send data  (...)
Close sockets  (...)
Total_Time = current_time() - t
```

Listing 10.2: Measuring latency on stream server.

To evaluate the overhead of sending messages and receiving message, the same measurements were done while sending an empty string. The reason for this experiment only going up to messages on 256 MB is because python started getting out of memory exceptions when sending messages on 512 MB. The server and the client was run on two different computers, both of the type seen in Table 10.2.

10.3 Local Tuple Space

When evaluating the local TS implementation, two different aspects were tested; the impact of tuple size and the impact of a populated TS (population is the number of tuples in the TS). Tuples with an approximate size from 128 KB up to 512 MB, doubling the size each step, were used to test each of the operations put, get, read, dump, and count. The tuples are of length 10, where the first element is

a string of size 128 KB to 512 MB, to get the desired size. The remaining elements are integers whose values are equal to its index in the tuple. The strings are generated initially in the program, before testing, and stored in a list.

For tuples of each size, the operations were repeated 1 000 times. This was also done for a TS with a population of 0 and up to 1 000 000, increasing the population in steps of 10 000. The tuples populating the TS are of length from 2 to 10 elements, where the first element always is one of the initially generated strings, and the remaining elements are integers whose value is equal to its index in the tuple. The TS is populated in a determinant manner, so each run will populate the TS with the same tuples. Some tuples are will be the same as the tuple used as argument for the operations.

The latency of each operation was logged by measuring the time difference from before and after the execution. The CPU usage and memory usage was logged for each put operation. The resident set size, virtual memory size, and the number of major page faults were not logged for this experiment. This was logged within the the TS, so it adds some extra overhead to the latency of put operations. The experiment was run on a computer with specifications as seen in Table 10.2.

10.4 Name Server

To test the performance of the NS a static NS was set up on a single computer (Table 10.1). 20 clients were set up on computers, both kinds the computer types where used. The clients simply send a request for the name server list 300 times. Note that the list will only contain a single element: the NS itself. The time of a request and the latency of the request is measure on the server. The latency of the request is the time from a request is received until the response is sent from the server. The client each log the time of any timeout.

10.5 Tuple Space Host and Tuple Space Client

The four experiments run to evaluate the TSH and TSC looked at these aspects:

- Impact of tuple size
- Impact of populated TS
- Impact of number of TSCs
- Impact of number of TSHs

In all these experiments a singe NS was used, running on a computer with the specifications as seen in Table 10.1. In the two first a single TSH and a single TSC was used. The same measurements were done on both TSCs and TSHs in all four experiments.

On the TSH the total processing time of a request was logged. This is the latency from a request is received until the response has been sent. The latency of receiving the request along and the latency of sending the result is measured, as well as the latency of serializing and deserializing these. The latency of calling the TS is also logged, and the resource usage is measured at the end of each request. The logging is done after the processing of the request is over.

On the TSC each operation is timed, and the resource usage is measured after an operation is finished. Each request also logs the latency of serializing the request, deserializing the result, and the total latency of calling the TSH and receiving a result, so the latter also includes all the processing time on the TSH. All logging except the total latency of the operation is logged within the operation call, so that adds some additional overhead to the total latency.

10.5.1 Tuple Size

When evaluating the impact of tuple size the same method as in Section 10.3 was used. A set of strings were generated and the tuples had a string as the first element, followed by 9 integers, whose values were equal to its index in the tuple. The only difference is that the size varies from 128 Kb up to 256 Mb. The reason for this is because python started getting out of memory exceptions when using tuples on 512 MB. This will be discussed further in Chapter 12. Each tuple is used as argument for the operations put, get, read, dump, and count, and each of these operations are repeated 100 times for each tuple.

Both TSH and TSC was run on a computer with the specifications as seen in Table 10.2. The TSH and TSC was run on two different computers.

10.5.2 Tuple Space Population

When evaluating the impact of a populated TS, the client generates a single tuple. First a string of 1024 bytes is generated and inserted as the first element in a tuple. The second element in the tuple is simply an integer with value 0. This tuple is then inserted 100 000 times, and then retrieved 100 000 times. The same experiment was repeated 10 times. The reason for only 10 repetitions of the experiment is because it is very time consuming.

Both TSH and TSC was run on a computer with the specifications as seen in Table 10.2. The TSH and TSC was run on two different computers.

10.5.3 Number of Tuple Space Clients

When evaluating the impact of the number of TSCs a single TSH was set up, while the number of TSCs was increased from 1 to 10. Each TSC inserts 1 000 tuples into the TS, where the tuple consist of a 1 KB string and an integer. Each TSC is passed a time up to 15 seconds, which is used to synchronize their actions. The TSH was run on a computer with the specifications as seen in Table 10.2. The TSCs were run on both experimental rigs. The TSCs were not run on the same computer as the TSH, but some TSCs ran on the same computers on high client numbers.

Due to conflicting results from the server and clients, the experiment was run a second time without logging on the server. Based on the results from both the first and second run of the experiment a slightly altered experiment was constructed. The logging on the server was changed so that it logs into different files for each client. To achieve this each client was assigned a unique integer as identifier, which was used as the second element in the tuple. This identifier was used on the server to log into different files. The number of TSCs was also increased from 1 to 30.

The results from this last experiment proved to be conflicting once again, so yet another change was done to the logging at the server. The request handler appends the total latency of requests to a list, which is written to file after the client closes the connection. The logging data from each client is written into separate files.

The final change to the experiment was to only log the total latency on both client and server, and these are logged by appending them to a list and writing it to disc at termination. This way there is no pollution within the measurements from timing operations and writing to file.

10.5.4 Number of Tuple Space Hosts

When evaluation the impact of the number of TSHs 30 TSCs were used, while the number of TSHs was increased from 1 to 10. Each TSC inserts 1 000 tuples into the TS, where the tuple consist of a 1 KB string and an integer. The integer is an unique identifier for the TSC inserting the tuple. To prevent pollution from logging data, only the total latency on the server and the total latency on the client was measured and logged. The latency is appended to a list and written to disc at termination.

Each TSC has a random change on insertion to terminate its current connection. This is done to better see the impact of the number of TSH. If this was not done, each TSC would just insert into a single TSH, and the load could get uneven. This introduces some extra overhead to the operations that must reconnect,

but it should not be that big. All TSHs ran on computers with specifications as seen in Table 10.1, while TSCs ran on computers with both specifications. Due to the number of TSHs and TSCs in this experiment, some TSCs were run on the same computers, all TSHs had at least one TSH on the same computer.

CHAPTER 11
EXPERIMENTAL RESULTS

11.1 STREAM SERVER

Figure 11.1 shows the cumulative frequency of the roundtrip latency and total latency for the client, as well as total latency for the server, when sending an empty string. The x-axis is the latency in milliseconds and the y-axis is the cumulative frequency in percentage. The total latency on the client is from a socket is created until it is closed, while on the server it is from a thread is spawned to handle a connection until the thread terminates.

Figure 11.2 shows the mean of the total latency on the server with standard deviation, and the mean of the total latency on the client. The total latency on the client is split into roundtrip latency and overhead, each with their respective standard deviations. The overhead is the total latency, subtracting the roundtrip latency measured for the same message. The y-axis is the mean latency in milliseconds.

Note that when measuring the total latency on the client, the first measurement was on 6.99 ms. Additional tests showed that the first point always had increased latency when executing connect. The increased latency in connect is due to the server host not being present in the address resolution protocol (ARP) cache on the client. Since it was not possible (due to limited privileges) to remove entries from the ARP cache, the impact it has on the latency could not be measured. Some additional tests were run, however, indicating that the additional latency is generally in the order of 1-2 ms. It is therefore likely that other aspects also contributed to the increased latency.

As it was not possible to find any additional causes to the extra latency, and the result could not be reproduced, except for the additional overhead from a miss in the ARP cache, the point has been removed from the set of results.

As seen in Figure 11.1 and Figure 11.2 the overhead from setting up a connection on both server and client is large compared to the roundtrip latency. The roundtrip takes less than 0.05 ms, while the total latency on the client is between

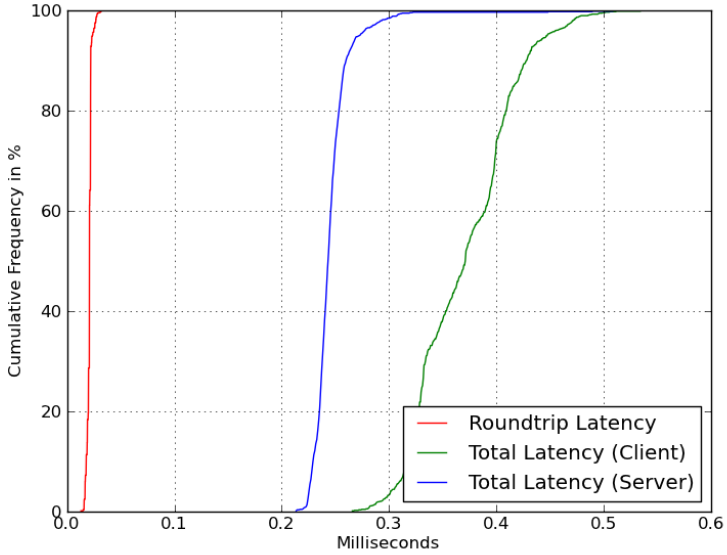

Figure 11.1: Cumulative frequency of roundtrip latency and total latency on client, and total latency on server.

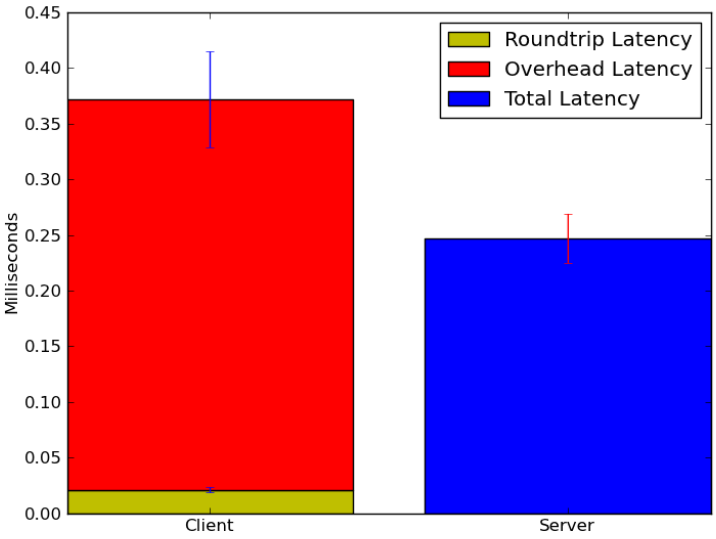

Figure 11.2: Mean value, along with standard deviation, of total latency when sending an empty string for both client and server. The total latency on the client is comprised of the mean value of the roundtrip and the mean value of the overhead, both with their respective standard deviations.

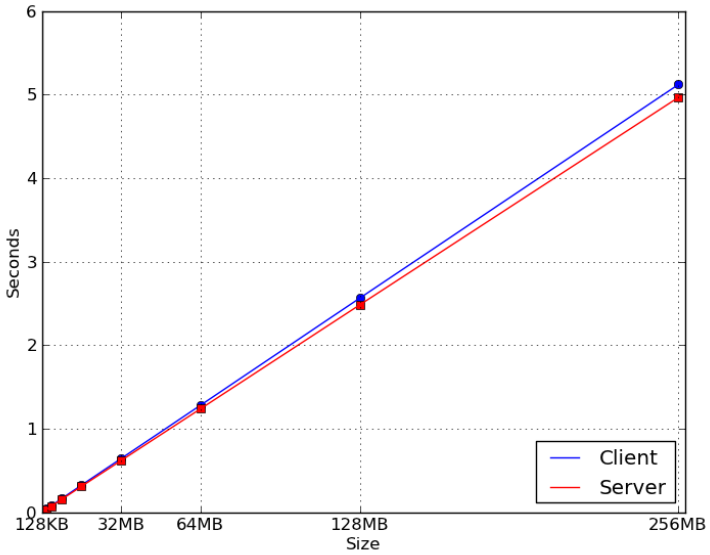

Figure 11.3: Total latency of message transmission for server and client as a function of message size.

0.3 ms and 0.4 ms. The total latency on the server is between 0.2 ms and 0.3 ms. We also see a much larger variation in the total latency on both server and client than we see in the roundtrip latency. It is important to note that the latency on the client includes most of the latency from the server, as it naturally has to wait for the server to respond when connecting.

Figure 11.3 shows the total latency, on both server and client, of sending messages of various size from client to server and back. The x-axis is the size of the message, while the y-axis is the latency in seconds. The standard deviation for these measurements can be seen in Table 11.1.

Figure 11.4 shows the average transmission rate for messages of different sizes. The x-axis is the message size and the y-axis is the calculated transmission rate in Mbit/s. The transmission rate is calculated by taking the size of the message, multiply it by 16, and divided it by the roundtrip latency. As size is in bytes, it is multiplied by 8 to get it in bits. Also, since the time is the roundtrip, the size of messages sent is twice the size of a single message.

Figure 11.3 shows that the latency scales linearly with the size of the message. The total latency on the server is also slightly lower than on the client, which becomes more clear for large messages. Figure 11.4 shows that the average transmission time for the smaller messages various from about 430 Mbit/s and up to 760 Mbit/s, while it stabilizes at above 750 Mbit/s.

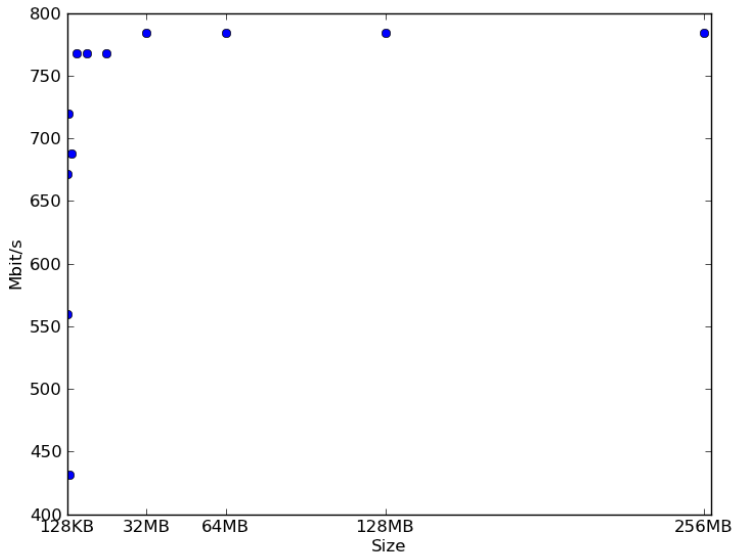

Figure 11.4: Transmission rate for messages of various sizes.

Size	Standard Deviation (sec)
128 KB	9.02×10^{-5}
256 KB	1.91×10^{-3}
512 KB	2.07×10^{-3}
1 MB	5.36×10^{-2}
2 MB	2.99×10^{-2}
4 MB	1.57×10^{-2}
8 MB	2.05×10^{-2}
16 MB	1.90×10^{-2}
32 MB	2.00×10^{-2}
64 MB	1.89×10^{-2}
128 MB	3.05×10^{-2}
256 MB	3.23×10^{-2}

Table 11.1: Standard deviation for total latency on the client.

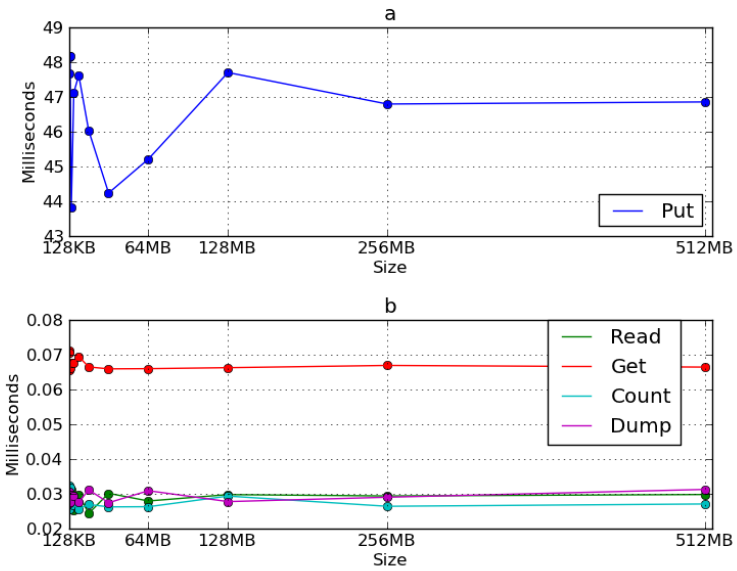

Figure 11.5: Latency of tuple space operations as a function of tuple size, with an empty tuple space. When performing all operations, except put, there is one tuple in the tuple space. When performing put, the tuple space is empty.

11.2 Local Tuple Space

Figure 11.5 shows the latency of TS operations as a function of tuple size. The x-axis is the size of the tuple used as argument for the operations and the y-axis is the latency of the operation in milliseconds. Figure 11.5a shows the put operation, while Figure 11.5b shows the other operations.

Figure 11.6 shows the latency of TS operations, and the CPU and memory usage, as a function of TS population. For Figure 11.6a The x-axis is the TS population at the time of the operation, the y-axis on the left is the latency of the operation in milliseconds, and the y-axis on the right is the resource usage in percentage. For Figure 11.6b the x-axis is the population and the y-axis is the operation latency in milliseconds. Figure 11.6a shows the put operation, while Figure 11.6b shows the other operations. The tuple used has size on 8 MB, but graphs using tuples of other sizes shows the same tendency.

As shown by Figure 11.5 there seems to be no correlation between latency and tuple size. The latency for put varies between 43 and 49 ms, and the only effect the tuple size seems to have is to stabilize the latency somewhat. The latency for get is very stable, being around 0.07 ms, while the latency for the remaining three operations all are about 0.03 ms. The variations in the operations seen in

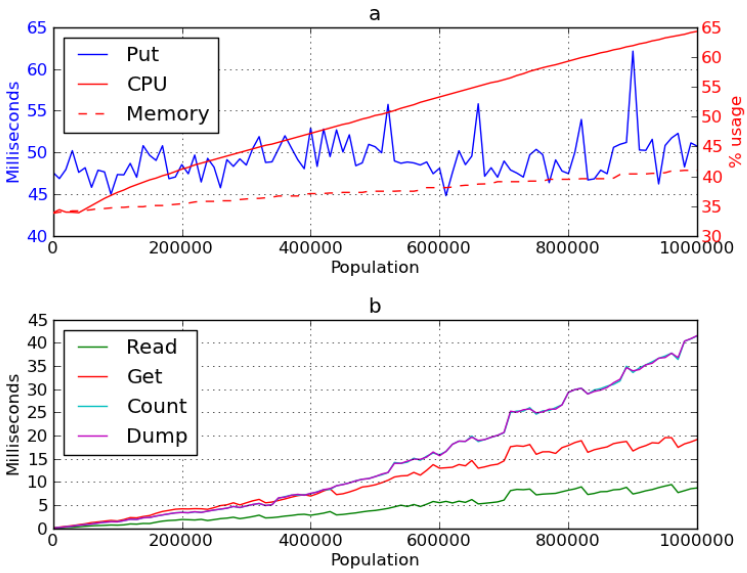

Figure 11.6: Latency of tuple space operations as a function of tuple space population. The size of the tuple used is 8 MB. Tuples in tuple space varies in size, from 128 KB to 512 MB.

Figure 11.5b shows the same tendency as for put; there is no additional latency as a result of the tuple size.

Figure 11.6a shows that there is no correlation between TS population and the latency of put. The latency varies from 45 ms and up to 60 ms. The memory usage increases from 33%, up to 43%, while the CPU usage increases from 33% and up to 64%. The low increase in memory usage is possibly because the logged memory is the physical memory that is allocated. It is possible that the process uses much more virtual memory, and that it swaps in and out from disc. The increase in CPU usage is probably because there is no limitation set; the process have to insert a lot of tuples as fast as possible, so it utilizes as much of the CPU as it can.

Figure 11.6b shows that the latency of read, get, dump, and count are dependent on TS population. Read has lower latency than get, and the difference gets higher at larger population. The reason the latency of get increases more due to population is likely because of increased latency in python when deleting entries in the dictionaries and sets used in the TS. Dump and count show even more latency than get, and they scale almost identical. They have similar latency because the only difference between them is that count returns the length of a list, while dump actually returns the list. Their increased latency compared to get is because

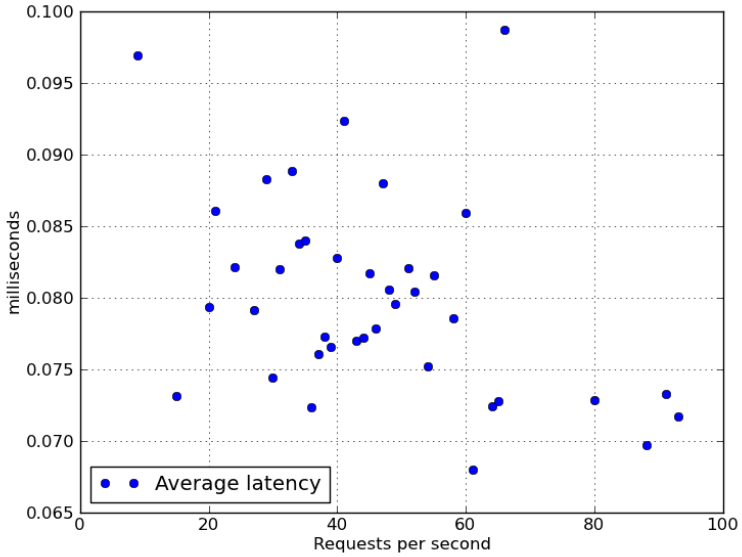

Figure 11.7: Average latency of name server requests as function of requests per seconds.

both operations have to retrieve *all* tuples matching the template, while get only have to return the first match.

Put should not be affected much by page faults, as it only work on data holding the indices of tuples, which should be small enough and used often enough to be located in memory. Get, read, count, and dump also mainly work on the index data. When they have a set of indices that may match the template, these have to be checked. Get and read only have to retrieve one and one tuple until one match. In this experiment the first tuple should match the template, but it can cause a page fault, as there are many copies of the same tuple, and the one that is being accessed may be located on disc. Count and dump, however, must access all of the tuples which might match the template, which makes it more likely that a page fault will occur. This is possibly a contributor to the increased latency as a result of population.

11.3 Name Server

Figure 11.7 shows the average latency of requests to the NS as a function of requests per seconds handled by the NS. The x-axis is requests per second and the y-axis is latency in milliseconds.

Figure 11.7 shows that the average latency varies between 0.065 ms and 0.1

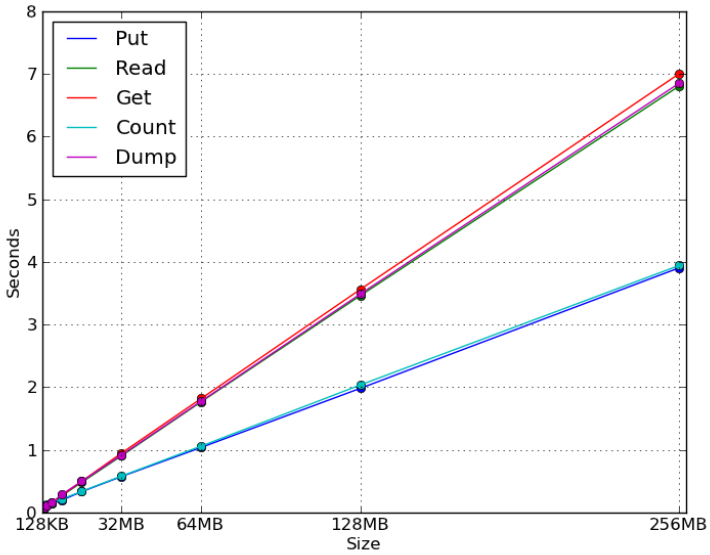

Figure 11.8: Latency of operations on shared tuple space as a function of tuple size.

ms. There is generally a larger latency for lower request loads, but most of the points are also located at lower request load. When running the experiment a big problem with the NS: if the NS receives enough requests per seconds it will spawn so many threads that it is not allowed to spawn new threads to handle incoming connections. This is because it spawns a new thread for each request. This also explains the results; the server can easily handle many requests per second with very low response latency, but it is limited by the amount of threads it can spawn. This causes it to peak at under 100 requests per second.

11.4 Tuple Space Host and Tuple Space Client

11.4.1 Tuple Size

Figure 11.8 shows the latency of operations on the shared tuple space as a function of tuple size. The x-axis is the size of the tuple used as argument and the y-axis is the latency in seconds.

Figure 11.9 shows the breakdown of each operation on the shared tuple space. The x-axis is the operation and the y-axis is the percentage of the total latency for that operation. Each operation is broken down into four main parts, which are given in their percentage of the total latency: the latency of serializing the

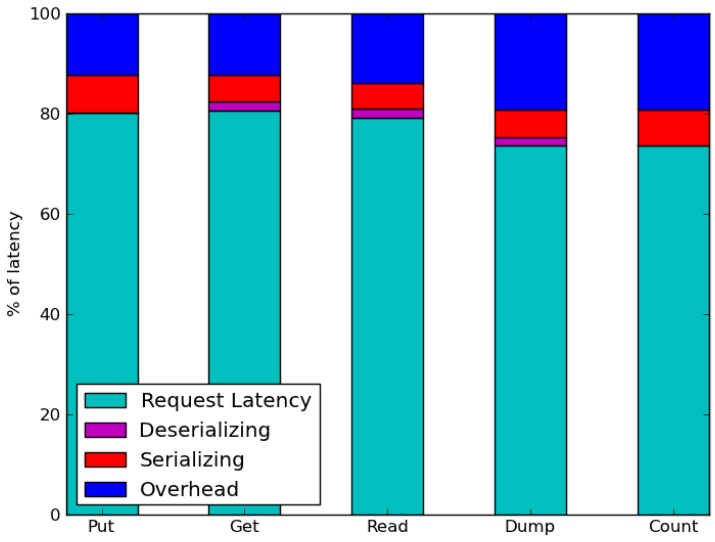

Figure 11.9: Breakdown of latency for each operation on the shared tuple space. The usage is represented in percentage of the total latency of the operation. The operations were performed for a tuple on 8 MB.

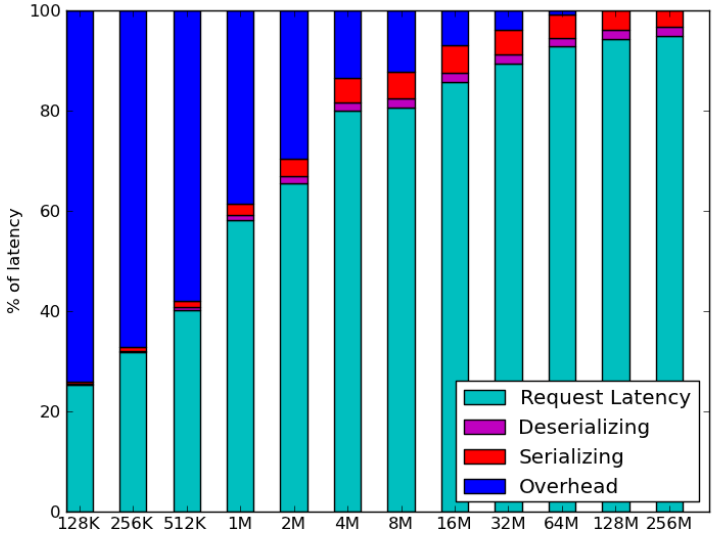

Figure 11.10: Breakdown of latency for the main components of the get operation for tuples of increasing size.

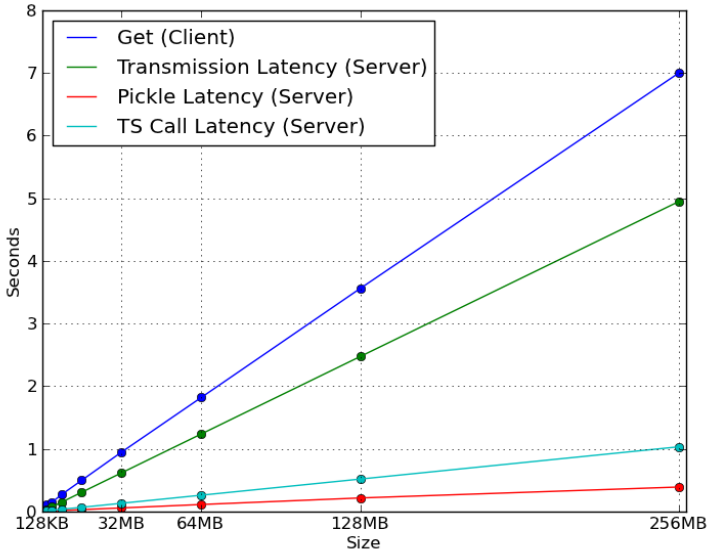

Figure 11.11: Latency of the get operations, along with the latency of the main components on the server, as a function of size. The transmission time is the total time the server used to receive data from the client and send the response back to the client. The pickle time is the time the server spend serializing or de-serializing data.

tuple, the latency of deserializing the response, the latency of sending a request to the server, and the overhead. Note that the latency of sending a request to the server includes sending the data to the server, waiting for the server to process the request, and receive the result from the server. The overhead is all time that does not fall in under the other categories.

Figure 11.10 shows the breakdown of the get operation on the shared tuple space for increasing tuple sizes. The x-axis is the size of the tuple used as argument and the y-axis is the percentage of the total latency. The operation is broken into the same categories as above.

Figure 11.11 shows the latency of the get operation, along with the latency of certain operations on the server, as a function of tuple size. The x-axis is the size of the tuple used as argument and the y-axis is the latency in seconds. The transmission time is the time the server spent receiving the request from the client and sending the result back to the client. The pickle time is the time the server spent serializing or de-serializing data. The TS call time is the time the server spent calling the tuple space.

Figure 11.8 shows that the latency of all operations increase linearly with the size of the tuple. The latency of put and count almost doubles as the tuple size doubles, while get, read, and dump increases more rapidly. The reason for lower latency for put and count is that the server returns much less data in case of success. While get and read returns a tuple, and dump returns a list of tuples, put only returns a small message signifying success, and count only returns an integer whose value is equal to the number of tuples in the tuple space at the server.

Figure 11.9 shows that, for all operations, most of the latency on the client is in sending the request to and receiving the result from the server. Put and count have so small deserializing latency that it is not visible. Count and dump show a slightly higher overhead, which is caused by these two operations having to aggregate results from all servers.

Figure 11.11 shows that, for this case, most of the latency when sending a request to the server is due to transmitting data over the net. 70% of the latency is due to transmitting data. There is also a increase in the latency of accessing the local TS as the tuple size increase. The transmission latency is of course linked to the size of the tuple, and as the tuple size decrease it is likely that the it will hit a point where the transmission time will no longer dominate the latency. This can be seen in Figure 11.10, where the dominant latency on lower tuple sizes is overhead on the client. As the tuple size increase, the latency for sending a request to the server dominates more and more, mainly due to the increase in transmission time. Serializing and deserializing also dominate more at larger tuple sizes.

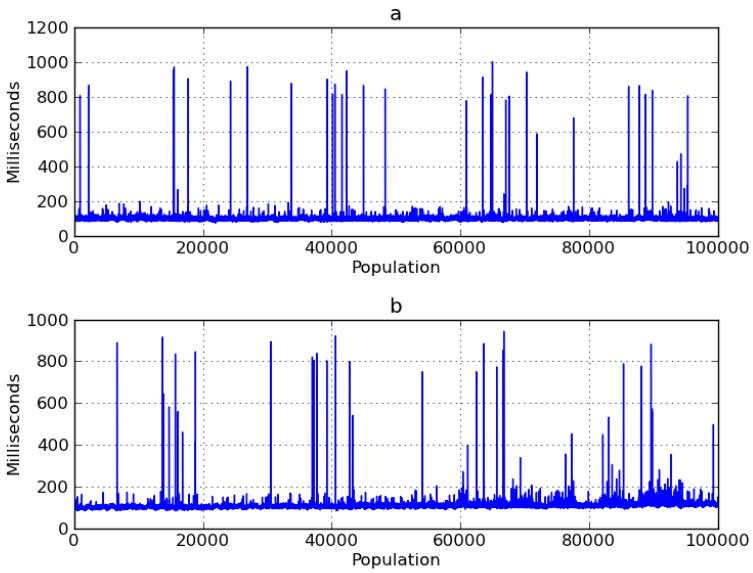

Figure 11.12: Latency of put and get on the shared tuple space as functions of tuple space population, measured on client. (a) shows the latency of put and (b) shows the latency of get.

11.4.2 Tuple Space Population

Figure 11.12a shows the latency of put, and Figure 11.12b shows the latency of get, on the shared tuple space, measured on the client. The x-axis is the TS population and the y-axis is the operation latency in milliseconds.

Figure 11.13 shows the cumulative frequency of put and get. The x-axis is the latency in milliseconds and the y-axis is the cumulative frequency in percent.

Figure 11.14a shows the latency of put, and Figure 11.14b shows the latency of get, on the shared tuple space, measured on the server. The x-axis is the TS population and the y-axis is the operation latency in milliseconds.

Both graphs in Figure 11.12 show that the operations have a fairly stable latency, varying mainly between 80 ms and 125 ms. There are, however, quite many peaks, going as high as more than 900 ms. There are more occurrences of peaks for put than for get. When studying the numbers more closely, the peaks are a result of extreme peaks in one of the ten runs. Some peaks are as high up as 8-9 seconds, which results in a 900 ms peak when taking the mean over all 10 iterations of the experiment.

As seen in Figure 11.14, there are only a few minor peaks for both put and get. Both operations shows an increase in latency, with the highest increase for get. Get also has a few spots where there is a sudden, significant, and continuous

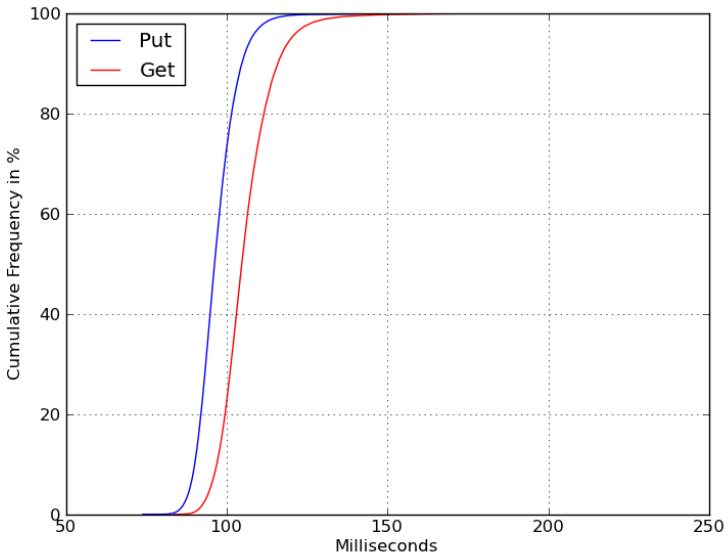

Figure 11.13: Cumulative frequency of latency of put and get on the shared tuple space with increasing population in the tuple space.

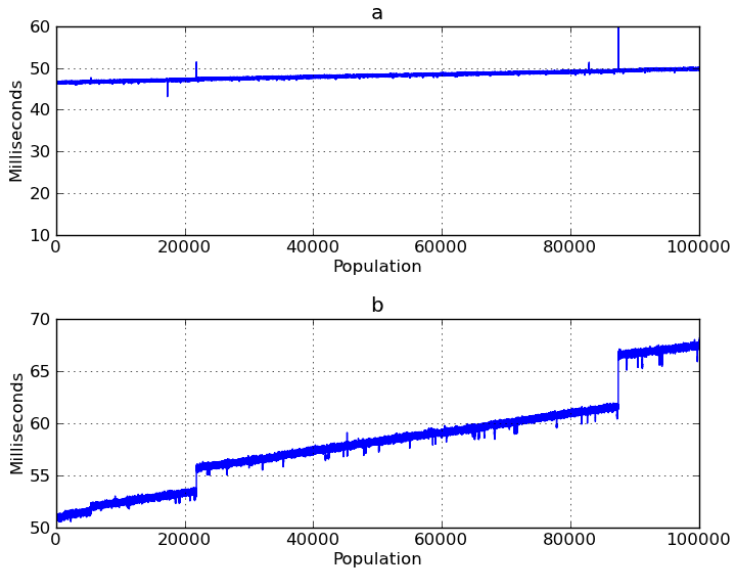

Figure 11.14: Latency of put and get on the shared tuple space as functions of tuple space population, measured on server. (a) shows the latency of put and (b) shows the latency of get.

increase in latency. These spots are possibly a result of caching; the first jump might be that the data no longer fits into the level 2 cache, while the second jump might be when the data no longer fits into the level 3 cache.

Figure 11.13 show that get have slightly higher latency than put. The reason for higher latency in get operations is that a successful put only have to return a small message signifying success, while a successful get has to return the tuple. This means there is more data to serialize and send on the server, and more data to de-serialize on the client.

The CPU and memory usage on the server is minor; the CPU usage increase from basically 0% up to 2.3%, while memory usage increase from 0.1% up to 3.9%. Over all the 10 iterations not a single major page fault was registered.

The cause for the peaks are hard to locate, as some are a result of increased latency when contacting the server, while most are because of increased latency elsewhere on the client. Additionally, when there is increased latency when contacting the server, there is not increased latency on the server, as can be seen from Figure 11.14. There are a few minor peaks, but they not even close to the peaks registered at the client. Since the increased latency cannot be traced in any of the log files, it is likely that the pollution is due to sudden increase in latency when logging at some points. Many of the peaks are close to each other, so the reason for the increased latency when logging might be because of other processes on the computer.

11.4.3 Number of Clients

Figure 11.15 shows the latency of the put operation using a single TSH, measured both on server and clients. The red line shows the total latency measure on the client, while the green line shows the latency of sending a request, getting it processes, and receiving the result from the TSH. The x-axis is the number of clients and the y-axis is the latency in seconds.

Figure 11.15 shows a significant increase in latency on the client side when the number of clients increase. However, the server does not see much increase in latency at all. While the clients experience somewhat linear increase in latency, the latency measured on the server remains almost constant, with a weak increase at the end. When comparing the request latency to the client latency, we see that the increase in latency is corresponding between these two measurements. The request latency measured consist of the latency to send the request, the latency for the server to process the request, receive the reply, and deserialize the reply. As the same request is sent all the time the latency of sending, receiving, and deserializing the request should be the same, and it certainly is not dependent on the number of clients in the system. Even so, the request latency shows the same increase as the total latency. This means the increase in latency is most likely due

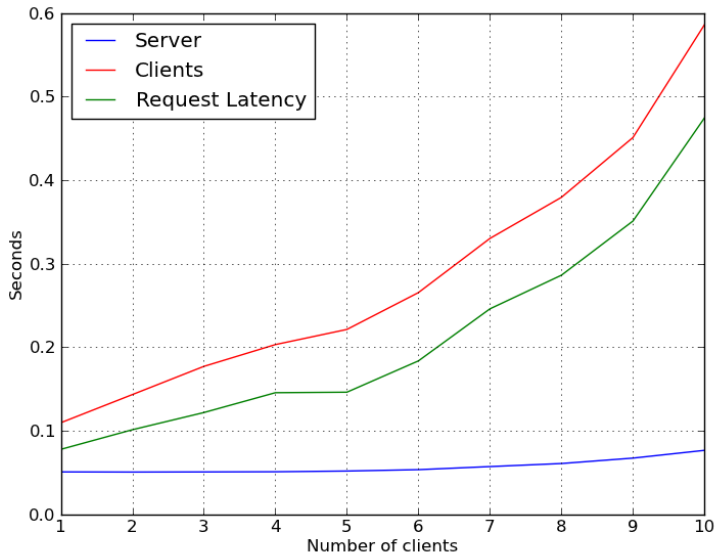

Figure 11.15: Latency of put operation on the shared tuple space as a function of number of clients, measured on both client and server. The request latency is the latency of sending a request, getting it processed, and receiving a result as measured on the client.

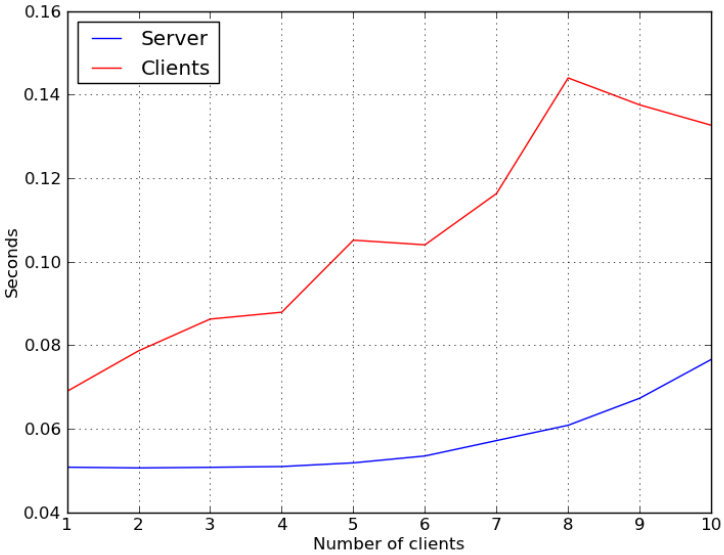

Figure 11.16: Latency of put operation on the shared tuple space as a function of number of clients, measured from clients and server in separate experiments. The numbers used for the server is the same as above, while the number for clients are from a second experiment where there was no logging on server.

to increased latency on the server which is outside the measurements.

Figure 11.16 shows the latency of the put operation using a single TSH, measured both on server and client. The measurements for the server are the same as seen in Figure 11.15, while the measurements for the clients are from a second run of the experiment where there is no logging on the server. The x-axis is the number of clients and the y-axis is the latency in seconds.

We see that the latency are closer in magnitude than before, with a sever decrease in latency on the clients. One possible cause for the pollution in the data is that the server logs data for each request into the same files. This causes the logging on the server to be serialized, so the request handlers block while waiting to get access to the logging files. This can be seen from the number in Figure 11.15; with one client the latency on the server is about 0.05 seconds and the latency on the client is about 0.1 seconds. On 10 clients the increase in latency on the clients is about 0.5 seconds, 10 times the latency on the server.

It should be noted that the clients wait at the start of a request, not the end. When the request handler on the server starts to log, the client finishes that request and sends a new one. However, since the client reuses the connection, it uses the same request handler, but that request handler is still waiting to finish logging

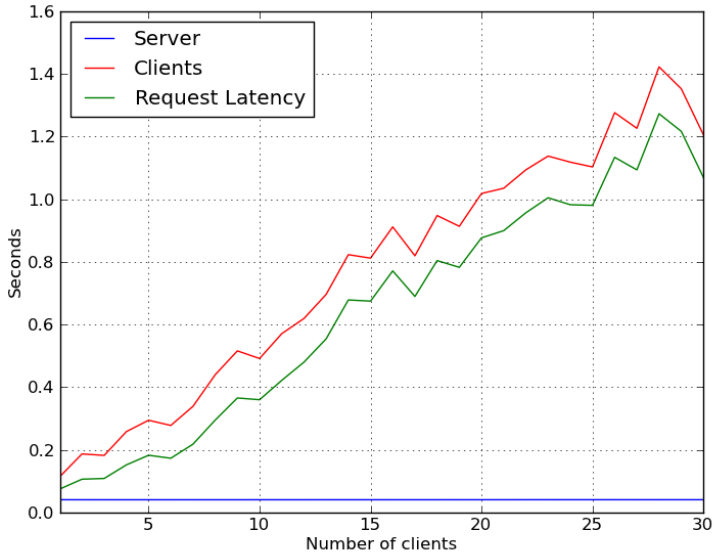

Figure 11.17: Latency of put operation on the shared tuple space as a function of number of clients, measured on both client and server.

data, so the client have to wait.

Figure 11.16 do show an increase in latency, but there is a small decrease in latency in the end. We also see a increase in latency at the server, although these numbers are polluted. To see the impact of number of clients, a new experiment was run, increasing the number of clients from 1 to 30, with logging on the server into separate files to avoid conflicts.

Figure 11.17 shows the latency of the put operation using a single TSH, measured both on server and clients. The x-axis is the number of clients and the y-axis is the latency in seconds.

As seen in Figure 11.17 the results are the same; there is a increase in latency on clients, but there is basically no increase at the server. Additionally, the server has a latency that is many magnitudes lower than the clients at 30 clients. The green line also shows that the client experiences increased latency on the server. Comparing the numbers seen in Figure 11.17 with those in Figure 11.15 shows that the latency measured on the clients seem to be the same. This points towards another cause for the increase in latency: the global interpreter lock (GIL). The Python GIL will be release when threads perform potentially blocking or long-running operations, such as I/O operations. The interpreter also tries to switch threads regularly, to emulate concurrency. It is possible that the threads have to acquire the GIL when opening and closing the log files and therefore still gets

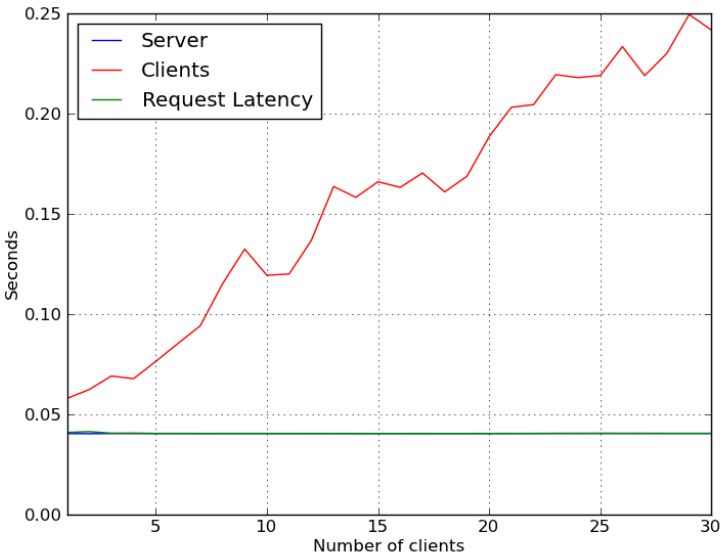

Figure 11.18: Latency of put operation on the shared tuple space as function of number of clients, measured on client and server. Limited logging on server.

blocked occasionally when logging. Additionally, if the threads release the GIL when writing to files, or the interpreter switches threads, there will be additional overhead.

Figure 11.18 and Figure 11.19 shows the latency of put operations on a single TSH, measured on server and client. The x-axis is number of clients and the y-axis is the latency in seconds. Figure 11.18 shows the total latency on the server, the total latency on the client, as well as the latency to get a request processed at the server. Figure 11.19 only shows the total latency on the server and the latency to get a request processed at the server.

We see from Figure 11.18 and Figure 11.19 that the client now experience the correct latency at the server, but that the total latency on the client still scales as the number of clients scale. The most likely reason for this result is the fact that the computers running the experiment runs NFS. This means that the I/O latency on one computer is actually affected by disc usage on other computers.

Figure 11.20 shows the latency of put operations on the shared TS as a function of number of clients. The total latency is measured and logged on server and clients. The x-axis is the number of clients and the y-axis is the latency in milliseconds.

In Figure 11.20 the results are finally what are expected. We see a significant increase in latency at the clients as the number of clients increase, from 40 ms up

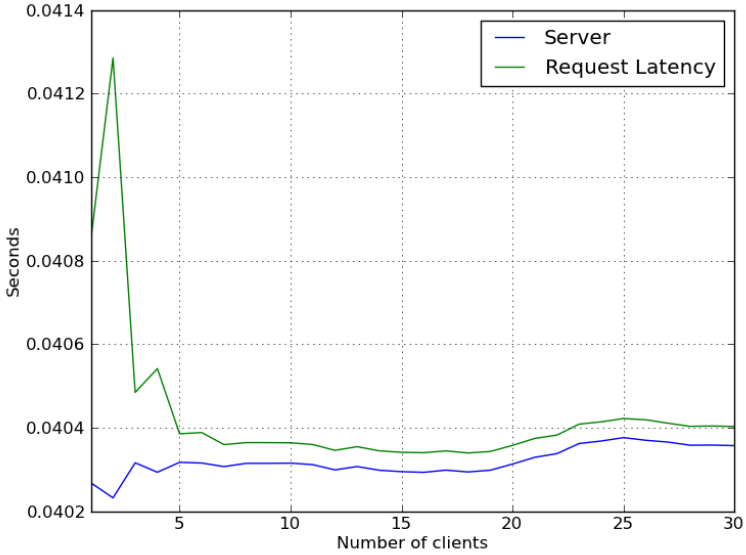

Figure 11.19: Latency of put operation on the shared tuple space as function of number of clients, measured on client and server. Does not show total latency on client. Limited logging on server.

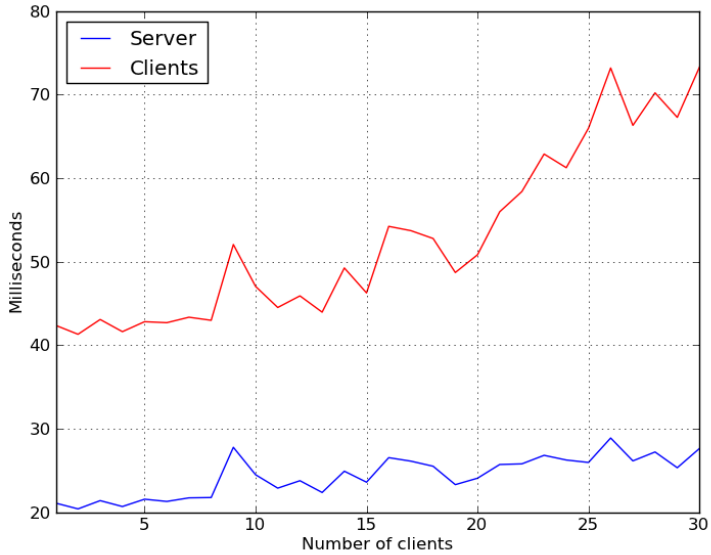

Figure 11.20: Latency of put operation on the shared tuple space as function of number of clients, measured on client and server. Only total latency is measured on both server and client.

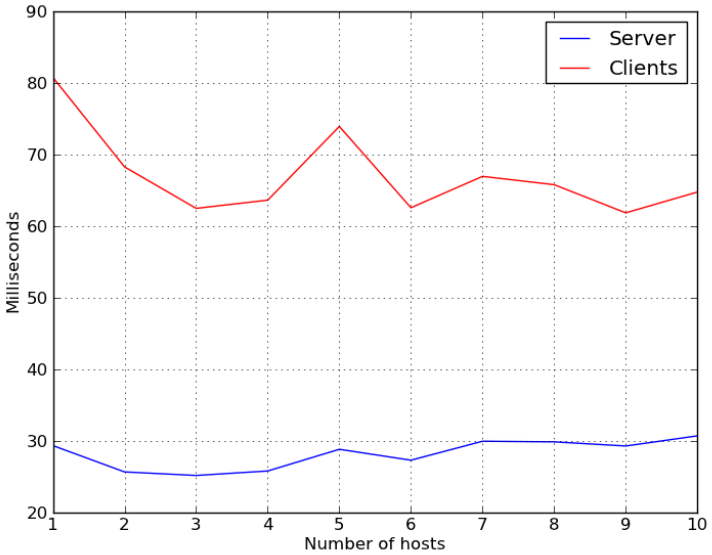

Figure 11.21: Latency of put operation on shared tuple space as function of number of hosts, measured on clients and servers.

to almost 80 ms. The server shows a small increase in latency, but not as big as seen on the clients.

11.4.4 Number of Hosts

Figure 11.21 shows the latency of put operations on the shared TS as a function of number of TSHs. The latency showed is the total latency measured on client and server. The x-axis is the number of hosts and the y-axis is the latency in milliseconds.

As seen in Figure 11.21 there is a drop in latency as the number of hosts increase up to 3. After 3 hosts, the latency is fairly low, except for a peak at 5 hosts. There are several possible solutions for the peak, such as others using the computers, or a majority of clients decided to use the same hosts a lot.

CHAPTER 12
DISCUSSION

12.1 TUPLE SPACE

This section will discuss the the aspects concerning the tuple space; the choice of local tuple space and why a shared tuple space was implemented instead of using an existing tuple space, experimental results, the architecture and design of the shared TS, and the implementation of the shared TS. Note that server and host will be used interchangeably.

12.1.1 LOCAL TUPLE SPACE

Searching for a suitable TS to use, with no prior knowledge on the area, proved to be difficult. Most of the TS implementations found was hard to use because of lacking documentation. Linuxtuples was supposed to be used in this thesis, it is a simple TS with good documentation on how to use it. Linuxtuples was used as the local TS on TSHs until the experiment phase, where it turned out to have a problem with exhausting file descriptors, as it uses TCP sockets to access the TS. As a result as simple TS implementation was done in Python, based on an unfinished TS implementation called Lindypy[1].

There are many more robust and versatile TS implementations, such as JavaSpaces, and many distributed TS implementations, such as Entangled[2]. Many of these are, however, commercial products, and most non-commercial TS implementations lack good documentation. The most promising TS implementation, LIME (see Section 4.2) was not used as it was found too late to be considered. It does have many features that are desired for the communication model, and it is possible it is a good alternative to the shared TS presented in this thesis.

[1] https://bitbucket.org/rfc1437/lindypy/src/2c0576344f5f/lindypy/TupleSpace.py (March 27, 2012)

[2] http://entangled.sourceforge.net/, (May 3, 2012)

12.1.2 Experimental Results

This section will discuss the experimental results for the shared TS.

UNDERLYING COMPONENTS

First we'll look at the underlying components in the system; the stream server and the local TS.

The results presented in Section 11.1 shows that, compared to the roundtrip of an empty package, the overhead in the SocketServer module is quite large, though the extra 0.5 ms latency introduced by the structure of the server is not extreme. As the transmission latency increase the overhead will become less significant. In a real case, the message sent will not be empty, and the network will have less bandwidth, meaning the transmission latency will be higher, and thus the overhead will have less impact.

In Section 11.2 we see the characteristics of the local TS used in the experiments. The TS used on TSH will have a big impact on the performance of the TSH, so having an inefficient TS as a basis can cause sever performance issues. The results show that the used local TS gets increased latency as the TS population increase, and the CPU usage increase steadily as the TS population increase. The experiments do test the system in un-realistic ways though; the system is not meant to handle neither tuples of size up to 512 MB, nor a population of 1 000 000 tuples. Put operations seem to be the most expensive one, having average latency of around 50 ms. The other operations can be said to have an average latency of up to 5 ms for realistic cases.

The performance of the underlying components are satisfactory. There is probably other TS implementation that can perform better, and the stream server used cannot, due to Python and GIL, give proper concurrent handling of connections. However, for this prototype they have good enough performance for their purposes.

One problem encountered when running experiments was that Python got out of memory exceptions when using large strings. When passing a string of 512 MB between a server and a client, the client would get this exception when receiving the string from the server. This was also a problem when using pickle to serialize a tuple with a 512 MB string. This actually gives an upper limit on the size of tuples the system can handle: 256 MB.

NAME SERVER

The experimental results show that the NS has sever performance issues. The NS has good response latency, but it cannot handle many connections per second. The server itself is fairly robots; it does not crash when an exception occurs. The

problem is, however, that it spawns a new thread for each request. When it has spawned too many threads, it is simply not allowed to spawn more. As a result, it cannot spawn new threads to handle incoming connection. It does not actually crash, but it still can't handle requests for some time. The server problems are discussed more later.

For the use in this thesis the NS has proved to be satisfactory, but in a real case the problems discovered during the experiments must be solved.

SHARED TUPLE SPACE PERFORMANCE

The experiments shows that the systems performance is dependent on tuple sizes, but not on the number of tuples in the shared TS. There is a slight increase in latency as the number of tuples increase, but this is so small compared to the total latency that it can, in most practical cases, be disregarded. The tuple size do affect the latency of operations significantly, as it is dependent on the bandwidth of the network. The transmission latency start dominating between tuples of size 512 KB to 1 MB. Before this the overhead in the system dominates. It is important to note that some of the overhead is introduced by measuring and logging the latency.

The results show that the latency of a put operation increase as the number of clients increase. The latency is fairly stable until the number of clients get up to about 7. After this point the latency starts increasing. For 30 clients the latency decrease as more TSHs are introduced, up to 3 TSHs. After this it is fairly stable. The clients in these two experiments sends many request to the TSHs, so the results indicate the performance of TSHs under heavy workload. The results indicate that a single TSH can handle up to 10 clients without much loss on high load. With a more normal load, a single TSH should be able to handle more clients.

The performance of the other operations as the number of hosts increase has not been tested in the experiments due to lack of time, but their latency is very dependent on the order the TSHs are contacted in. Assuming we have a system with 30 clients and 10 TSHs, a get operation to a single TSH should have slightly higher latency than put, so we assume it to be 70 ms. If the first TSH can provide a matching tuple, the latency of the operation is 70 ms. However, if the request must propagate through all TSHs, returning a matching tuple from the last TSH contacted, then the latency of the same operation will suddenly be 700 ms. Read will have similar performance. Of course, the performance of a request to a single TSH is dependent on many different aspects, such as the TS implementation used and the number of tuples stored at that TSH, and the workload on the TSH.

The performance of dump and count will decrease as the number of TSHs increase, because they *have* to contact all TSHs.

Even though many of the numbers from the experiments are a bit high due to pollution from measuring and logging, these numbers are probably lower than they would be in a real case. In the environment the system is designed to be used, the network bandwidth is likely to be many magnitudes lower, and the hardware on devices hosting TSHs and TSCs will probably be worse. It is very likely that the hardware running on the drones will, in many cases, be a lot worse than the hardware used in the experiments. One possibly problematic aspect is serialization of messages. When serializing a message on the experiment rigs, the latency was acceptable for various size. For sizes such as 256 MB, the latency was generally as low as 0.2 seconds. However, when serializing a message on 256 MB on a virtual machine, run on a computer with lower specifications, the latency was generally around 5 seconds. It is reasonable to assume that some robots that could use this communication model will have even worse performance than this.

There are also two quite important aspects that are not considered at all in this thesis; security and power usage. A drone, and possibly some devices hosting TSHs, will be powered by batteries, so power consumption is a problem, which can put constraints on the system. These constraints can affect the performance of the system. Also, it is important that such a system has some sort of security, which probably also will affect the performance.

For tuples of size 1 KB the latency experience by the client is generally in the range of 50 to 100 ms. The latency should be low enough to be used as a communication model for drones, but it probably is not viable for messages that need low latency. E.g. data that is being used in an interactive manner, such as a video stream used for manually flying the drone, should not go via a shared TS. If the shared TS was tested in a real case, the architecture and design can be refined to perform better, and the implementation can be reworked in a more efficient programming language, giving less overhead.

The use of a shared TS is naturally dependent on the application domain; in some cases the latency of 50 to 100 ms is acceptable, while in other cases it is not. One of the important motivations for a shared TS is the temporal uncoupled message passing; it is conjectured that it is desired to be able to send a message that will be retrieved at a later time. These messages are often not dependent on extremely low latency, its enough that some process will retrieve it at some time, and that the latency is low enough that the message can be retrieved before a disconnection happens.

As mentioned earlier in the thesis, it is probable that message that are dependent on latency should be passed over a more direct communication link. A shared TS should probably be used mostly for messages that are not dependent on being retrieved fast, and also messages that have no specific destination. Urgent messages requiring low latency are likely to be passed between two devices that are already aware of each other, making it feasible to set up a direct link between

them.

12.1.3 Architecture and Design

The TS consist of two types of participants: hosts (or server) and clients. A host provides a service for clients; a TS where messages can be inserted and retrieved from. The client in the system which is discussed in this thesis should be assumed to be highly mobile. This causes their environment to be one where disconnections should be expected, and it should not be assumed that it can be within range of a specific network. It should also be expected that the communication link between client and host can have low bandwidth.

ALTERNATIVE ARCHITECTURES

Considering these features of clients, it is undesirable to have a centralized structure to the system, where a single server hosts the TS. This would have a single point of failure; if the centralized server goes down, the whole communication model goes down. More importantly, it would force clients to a have a single point to communicate through, which would probably be unavailable often. It would also severely limit the area in which clients could communicate with each other.

It is therefore desired to have a decentralized structure to the TS; the TS should be distributed over several hosts. This makes the likelihood of failure decrease, as there are several hosts who provide the service, and it allows for having hosts located at different physical locations, and have host located in different networks.

The host should be able to move, but it is likely that they will not be as mobile as clients. It should still be expected that hosts will experience both disconnections from clients and disconnections from other hosts, and it should be expected that two arbitrary hosts may be located in two disconnected networks for a long period of time.

When designing a decentralized system there are several possible architectures. One possibility is a peer-to-peer (P2P) network, where all participant in system are equal and operate as both servers and clients. P2P networks are often divided into two categories: structured and unstructured. Structured P2P networks have a deterministic procedure for creating an overlay network, usually based on distributed hash table (DHT), while unstructured P2P networks create the overlay network in a basically random manner.

One main problem with using a P2P architecture is that it should not be assumed that all participants in the system can act as both server and client; many robots will likely have limited processing power and storage, so they cannot spend

their resources on hosting a TS. Another problem relates to the structure of the P2P system.

In a structured P2P network the whole overlay network is restructured when nodes join or leave. In Chord[33] nodes are organized in a ring. When a node joins, it generates a random identifier *id*, which determines its location in the ring. When a data item is inserted with a key *k*, it is mapped to the node with the smallest identifier *id* such that $id > k$, which is called the successor of k, or succ(k). So when a node joins the network it takes over the responsibility of all items in the ring that is mapped to it. When it leaves it transfers all data items to its successor.

First of all, transferring data item when joining an leaving causes a lot of unwanted network traffic, especially in cases where there is limited bandwidth, and many participants will be joining and leaving the system a lot. Second, in most cases nodes will not leave the system in a graceful manner; they will most likely move out of the network range and be disconnected, leaving no time to leave the overlay network. Third, the use of wildcards when looking up tuples will probably complicate lookups a lot, as they cannot simply be mapped to a specific node.

In a unstructured P2P network these two problems are not present, but, due to the lack of structure, the only way to locate a specific data item is to flood the network with the query, again leading to unwanted network traffic.

APPLIED ARCHITECTURE

The architecture of the system was chosen to be decentralized servers; it keeps the client-server model, but there are many servers instead of only a single server. The hosts do not communicate with each others, which is a result of too little time to handle the added complexity this would cause. As a result there is no load-balancing between hosts. Each host works completely independent from all other hosts, and the logic to build up a single tuple space from all the fragmented tuple spaces located at separate hosts is located at the client. The client has to contact one host after the other, until it has fulfilled its own request. It can therefore be argued that it is not really a distributed system. There are many definitions of a distributed system, one which is given by Andrew Tanenbaum and Maarten Van Steen[34]:

> ”A distributed system is a collection of independent computers that appears to its users as a single coherent system.”

They specify that to appear as a single system, there must be some collaboration between the independent computers. To avoid any confusion, the system is called a *shared* tuple space, or shared TS.

Clients need a way to locate hosts to be able to contact them. Initially the idea was to use multicast to broadcast a request for the address of hosts, but this causes a lot of excess network traffic, and it limits the availability of the shared TS to clients that have access to the multicast domain. Name servers are used to solve this problem. Two types of NSs are introduced: static and dynamic. A static NS is assumed to have a static address, while dynamic NSs are assumed to be executed by hosts, so they might be mobile and thus change address. Dynamic NSs use multicast, thus enabling clients to locate them through a multicast domain, making it possible to use the shared TS without any prior knowledge of any other device. The ability of hosts to start dynamic NSs on demand also ensures the existence of at least one NS on a network with hosts presents, assuming at least one host is working as intended. Static NSs allows clients to access hosts located on another subnetwork. The use of NSs limits the network traffic and adds an easy way of locating hosts.

Splitting the main components of the shared TS into host and client gives a clear distinction between actually being a part of the shared TS and only accessing it. A process can run a host to participate in the shared TS, and it can run several clients to access it. Having a lightweight client also removes a lot of excess load introduced by having no distinction between the two, which is a positive aspect for low resource devices. The ability to run several clients is also beneficial as it gives programmers more freedom, as well as the possibility of concurrent access to the shared TS. Instead of passing a reference to client between objects, each object can potentially have their own TS client.

Architectural Strengths and Weaknesses

The main strength of the architecture is the limited complications related to disconnections. If a host is disconnected from other hosts, there is no change, as there was no communication between them initially. If a host is disconnected from a client, there are mainly two complications; if they were communicating at the point of disconnection, the connection is broken, which must be handled at both ends, and the client will lose access to the tuples located at the host in question. The low complexity of disconnections makes it easier to handle devices moving in and out of networks.

The combination of low complexity for disconnections and dynamic creation and termination of NS on demand gives a big benefit; it gives very seamless transitions between a single shared TS and many separate shared TS. If we have a shared TS consisting of 10 hosts, and these are split into three separate networks, the hosts spawn new NSs on demand, giving a smooth transition over to three separate shared TSs on three different networks. If these are then moved into the same network again, the excess NSs should be terminated, and the remaining NSs

are made aware of the new hosts in the network, again giving a smooth transition back to a single shared TS.

Losing access to the tuples at hosts that become unavailable can be seen as a weakness, but as it cannot be assumed that a host is always available, it can not be assumed that a previously inserted tuple will be available at an arbitrary point of time. Losing access to tuples is something that must be expected and handled no matter the architecture.

One weakness of the architecture is the load it places on the client. As mentioned, it should be assumed that some clients have limited resources. It is also likely that hosts have as good or better communication links available as clients. With this in mind, it might have been a better solution to set up the shared TS as a distributed TS; the clients only contact a single host, which then contacts other hosts and produces a result for the request. This relieves the clients of some work as well as decreases their network traffic.

DESIGN

Tuple Space Host

The main motivation behind the design is high compatibility. Many implementations of TSs support direct remote access to the TS, but this requires knowledge on the client side of the specific interface used by the TS. Using a TCP/IP server to access the TS allows TSCs to access TSHs, running any type of local TS, with the same API; the TS used on the TSH is transparent for the TSCs. It would also allow clients to contact TSHs using any programming language if the serialization used wasn't language specific. However, as the serialization method used is pickle, which is Python specific, contacting TSHs using other programming languages gets more complicated.

Splitting the access logic to the local tuple space into two layers is to simplify incorporation of different TS implementations. Default operations are in place to protect access to the TS and access it. So when incorporating a new TS implementation, the startup and termination of the TS must be implemented. In many cases the functions to access the TS must also be implemented, as the interface between TSs will probably vary. In those cases, a the same synchronization mechanism can be used, and the interface is already specified, so the functions only need to be overridden.

Tuple Space Client

The reason for having a separate thread to handle insertion of tuples is to ensure that a tuple will be inserted at some point, provided that the process does not terminate. Without this a process could try to insert the tuple, but fail, which in

some cases can be unfortunate. This is mainly a problem when there are no TSHs within range. It is still possible that the tuple will not be inserted, such as if the process terminates before the TSC can reach a TSH.

The choice of keeping the connection open for some time was done after running more than 50 TSCs on a cluster behind a Network Address Translation (NAT). Since the TSCs could send quite many requests per second, the number of requests quickly added up. Each connection required a entry in the NATs routing table, which added up to a total of 65 000 entries quickly, as the NAT had to keep the entries for some time. This caused the whole cluster to become unavailable, as new connections were not allowed due to no free space in the NATs routing table.

Keeping the connection open solves this problem, as well as saves some overhead. As seen in Figure 11.1, the overhead of setting up a connection is quite expensive, between 0.2 to 0.4 ms. It is not much overhead, but it can quickly add up in the long run. Looking at Figure 11.13, we see that the latency for inserting a 1 KB tuple into the shared TS, in most cases, take less than 100 ms. This means that the overhead of creating a connection would count for roughly 0.4% of the total time. This number will increase as the size of the tuple decrease, and it is not unlikely that the tuple size will generally be much lower than 1 KB. However, it is also likely that the bandwidth of the network used will be significantly lower than the network used in these experiments. This will make the overhead less significant. Re-using sockets might have other positive effects too; the TSC might need to contact NSs less, removing some load from them and the additional overhead introduced by this.

The requirement for the TSC to send an acknowledgement is to handle disconnections. Of the current five operations, only two of them actually modifies the shared TS: put and get. So for these two operations there must be some policy for what to do in case of disconnection in the middle of a request. Lets look at three different scenarios where the TSC accesses the shared TS and a disconnection occurs; a disconnection before the operation is passed to the local TS, a disconnection after the operations has been performed in the local TS, but before the result can be sent successfully to the TSC, and a disconnection after the result is sent to the TSC but before the acknowledgement is returned.

For read, dump, and count there neither of these needs any special handling. In the two first cases the TSC will experience that the operation on that TSH failed, while in the third case the TSC will experience the disconnection, but as the result is received, it will regard it as a success.

For put and get a disconnection before the request is passed to the local TS at the TSH will be a failure, and no action need to be taken on the TSH, as the TS was not modified. However, if the disconnection happens after the TS is accessed, the TS has been modified. In this case the TSH will does not know if whether the TSC received the result or not. The policy is that it is more important to not lose

tuples, so if a put operation was performed, the tuple is not removed, while if it was a get operation, the tuple is re-inserted into the TS. If the TSC did not receive the result of a put operation it will be performed again, and there will be duplicate tuples. If the operation was a get, there will be no change as the tuple is reinserted. If the TSC received the result, but the TSH did not receive the acknowledgement, a put operation will not be performed again. However, in the case of a get, the TSC will receive the tuple, but the tuple will still be present in the TS, giving duplicate tuples.

As said, the policy is that it is more important to not lose tuples than it is to not have duplicate tuples. Assuming a critical task is posted using the shared TS, the one posting the task should be guaranteed that it should not get lost. It is better that the task is performed twice than that the operator posts it and turns her attention elsewhere, only to realize later that the tuple was lost and therefore the task was never performed.

Tuples can still be lost however, such as if a TSH terminates, as there is no mechanism for transferring tuples to other TSHs or storing tuples in a persistent manner. Another way to lose tuples is if the tuple is still in the TSCs queue, but has not been inserted yet. The client will see each put operation as a success right away, as the tuple is simply added to the TSCs FIFO queue, but the tuple might not be inserted into the shared TS for some time, especially if there are no TSHs available at that time.

12.1.4 Name Server

As the NS is a very small part of the system, and the design suffers somewhat from being down-prioritized. It was designed and implemented to work and provide its service. Although it is a small part, it is a integral part, without it the whole system will break down. As it is based on the SocketServer module, it is fairly robust, which is important, but it has performance issues, as seen in Section 11.3, mainly due to spawning too many threads. The lack of performance is not necessarily a big problem; the system is designed to support several NSs, and the load on the NSs should be fairly low. The components in the system are designed to not contact the NS too often, so assuming a component contact a NS once every two seconds, and each device runs two components (e.g. one TSH and one TSC), then each device will contact the NS once every second. A single NS should then be able to handle 40 devices alone, so having three NSs with good load-balancing between them lets the system handle 120 devices, more than what it is meant for at this point.

Even though the current design and implementation of the NS is sufficient for the current needs, the design and thus the implementation should both be drastically changed. It is essential for the system to have NSs that work properly,

and having NSs that potentially breaks down if they receive too many requests, as well as the poor performance, is bad. The new design should focus on increasing performance somewhat, but mostly on improving the robustness of the NS. Alternatives are to use a pool of threads to handle requests, instead of spawning and terminating threads all the time, or to use a HTTP server.

The reason for having dynamic NSs is to enable TSCs to locate TSHs no matter what, as long as there are TSHs in the network. If a TSH notices that there are no NSs in the network, it spawns a new NS. There will be a small delay between there being no NSs in the network until the TSHs start noticing, in which a TSC can only contact any TSHs it already knows about. The reason behind static NS are twofold; first, it lets the system have more stable and reliable providers of the NS service which is available from the start, and second, it allows access to resources outside the subnetwork. As the dynamic NSs can only be detected on by devices on the subnetwork, resources outside the subnetwork cannot find these. Static NSs allow remote resources to have an access point to the system.

It could be a useful to simply always have a NS running on the same device as TSHs, and use gossiping to spread information about TSHs. However, this could introduce extra network traffic, which might not be desired.

12.1.5 Implementation

TUPLE SPACE HOST

Even though all experiments have been run using SimpleTS as the local TS on TSHs, Linuxtuples is still available, and code is still available to use Linuxtuples. The reason for changing from Linuxtuples was simply due to a problem with handling file descriptors in Linuxtuples; some file descriptors, most likely TCP sockets, were not released properly, causing all file descriptors to be exhausted and the whole system breaking down. As a result an alternative had to be implemented. As this problem was discovered late, there was not much time to spend on implementing a proper alternative. Fortunately Lindypy[3] was found, and this was used as the basis for the TS implementation. The reason for not simply using Lindypy is threefold; first, it did not support wildcards, only formal parameters, and it did not support count or dump, which would cause conflicts with the whole design of the system. Implementing an own TS using Lindypy as a base allowed for using customizing it to have basically the same interface as Linuxtuples. Second, using Lindypy would require all devices using the system to actually download and install the module. Third, the implementation of Lindypy is much more complex than what is actually needed.

[3] https://bitbucket.org/rfc1437/lindypy/src/2c0576344f5f/lindypy/TupleSpace.py (March 27, 2012)

TUPLE SPACE CLIENT

The reason for splitting the TSC into two classes was to remove methods from the TSC class unless they needed to be there. As the TSC is used as an interface to the shared TS, it should really only hold methods that are supposed to be called by the client, so the extra threads and the helper methods for inserting tuples should really be moved out of the TSC class.

There is also one problem related with inserting tuples and not closing the connection right away after contacting a TSH. Since insertion of tuples happen in a separate thread, it is possible for two operations, a put and either a read, get, count, or dump, to happen at the same time. As they actually use the same socket, they might get blocked when trying to contact a TSH, causing extra latency on one or both of the operations. To solve this problem the put operation could have its own socket, or two sockets might be set up, and the operations just use one which is free.

NAME SERVER

As mentioned above, the design and implementation of the NSs is not as good as it should be. It is based on the SocketServer module, and uses multithreading, which causes it to break down if it receives too many requests per second. An attempt was made to use a single-threaded server, which solved the problem of the server breaking down, but it caused a lot of other problems, as clients timed out since the NS could not handle the requests fast enough.

12.2 Robot Control Unit

This section will discuss the architecture, design, and implementation of the RCU, and discuss the problems encountered. Due to limited general knowledge on the area of robotics, there will not be a discussion about alternative architectures and designs.

12.2.1 Architecture

The approach was to have a generalized system that do not place too much limitations on the type of robot and its sensor. The same approach as with the local TS was used, the RCU consist of two components; a main controller which is dependent on the class of robot (e.g. plane vs. helicopter), and a robot controller which handles incorporating the specific robot with the rest of the system. Using this approach can remove some workload when incorporating new robots with the

system, as similar robots might be able to use the same code, but it also forces a pre-defined structure on the programmer.

12.2.2 Design

As the only difference in the main controller from different classes of robots is the features of the robot, it is mainly reusable.

The robot controller has also been split up into two components: the universal controller and the specific controller. The universal controller provides code that must be present in all robot controllers, while the specific controller is mainly dependent on the brand and model of the robot, and must therefore be implemented specifically, although it should be possible to reuse much code between similar robot classes.

This design might make it easier to incorporate new robots, but it might also just be a rigid structure causing frustration. As only one type of robot was used, it is difficult to evaluate if the chosen architecture and design really makes incorporation easier or not. It is very possible that the proposed design is simply bad, and that it fits better as a frame of reference for how to incorporate robots in the system.

12.2.3 Implementation and Problems

A very big part of the implementation revolved around image processing. A set of functions were created to process images, and these functions were utilized in another set of function, which perform image analysis. These functions were created to be used by any type of robot, to give tools for analyzing video streams. The image analysis functions were again used to implement methods for controlling the AR.Drone autonomously by using the video stream.

Unfortunately the AR.Drone provides very little useful data. The sensor data provided by the drone, except for video stream, is velocity along all axes and direction. However, the velocity proved to be too inaccurate to be useful; the velocity along the z-axis is always set to 0, while the velocity along the other two axes was very inaccurate, on the verge of appearing as random. The IGS also doesn't report any velocity unless the engines are running, making it harder to debug and understand what is going on. Also, the direction reported is only an approximate offset from the direction as the drone receives power. When the battery is connected to the drone, the current direction of the drone is set to 0.

The task to follow an object with a given colour was implemented to learn some image processing and image analysis. It was meant to be expanded to popping balloons, which could have been used to test the incorporation of robots in the system. However, two problems presented itself. The balloons were attached to a

wall, but the drone had problems flying so close to the wall, and usually flipped over. The second problem was that when the robot got close to the balloon, the camera detected too little light and automatically adjusted the brightness and contrast levels. This caused a previously red balloon to no longer be recognized as red, giving problems with following it and detecting if the balloon disappeared or not.

The second attempt to come up with an autonomous task that could be used to test the incorporation of robots in the system was to make the drone fly a route autonomously. Markers were supposed to be placed on the ground, helping the drone to adjust its heading and estimate its position. The markers had to be placed in the same direction, so the drone could adjust its estimate of its position whenever it found a marker. The velocity data from the IGS and the estimations from the markers could be used to build up a virtual map, which the drone could use to navigate.

The inaccurate of the velocity data from the IGS, and the fact that the drone drifts too much in flight, made it hard to accomplish this. Since the drone drifts too much, it loses its way when flying between markers, and since the velocity data cannot be used to negate the drifting, the drone would end up crashing into a wall.

In the end inexperience within the fields of robotics, image analysis, and image processing, along with limited sensor data from the drone, made it too time consuming to accomplish anything with the robots. As a result the focus of the thesis was shifted from the system as a whole over to the communication model.

CHAPTER 13
CONCLUDING REMARKS

This thesis discusses a software platform for semi-autonomous robots, and presents a shared tuple space designed as a communication model for robots in a mobile environment. The shared TS offers a way of communicating in a spatial and temporal uncoupled manner, removing much of the complexity related with communication in a mobile environment. The contributions from this thesis are:

- An architecture and design for a transitory and shared TS, designed for a segmented network with devices moving between networks.
- A mechanism for detecting devices hosting a part of a shared TS.
- A realization of the architecture and design, implemented in Python
- Experimental results characterizing the shared TS.

The shared TS has been evaluated by running a set of experiments, measuring its performance under different circumstances. Although the shared TS has not been used to pass messages between devices in a mobile environment, the experiments have indicated that the performance of a shared TS is satisfactory. One of the motivational aspects for using a shared TS was to get temporal uncoupling, so that a message can be passed without considering whether or not the recipient is available at the time or not. As the shared TS is mainly designed for messages that do not necessarily have to be retrieved by any device as fast as possible, it is not extremely dependent on low latency.

13.1 LESSONS LEARNED

The experiments show that the latency of communication models are highly dependent on the bandwidth of the network; often the overhead of the communication model used is insignificant compared to the latency of simply transmitting

a message between devices. This becomes even more relevant in a mobile environment, where the network bandwidth can vary a lot, as a robust and efficient infrastructure is not necessarily available.

One of the big challenges in a semi-autonomous robot platform is simply passing messages between devices, as it cannot be assumed that the receiving device is available for the sender, and it can be hard to locate the recipient in the network. Messages that are dependent on low latency can often be sent over a direct communication link, but the problem here is to simply locate the device. As such, it is not necessarily important to focus a lot on the latency of the communication model; the service provided by the communication model such as spatial and temporal uncoupled communication, and possibly offering a simple way of detecting other devices, can often be much more important than low latency.

CHAPTER 14
FUTURE WORK

This chapter looks at some aspects related to the work in this thesis that could be researched further in the future.

One natural direction is to integrate a shared TS as a communication model with a semi-autonomous robot system, evaluating its viability for this case. Experiments should be run to see how a shared TS performs in a mobile environment, such as latency, handling segmentations of the shared TS and the network, disconnections while communicating, and device detection. The communication model should also be used to actually control robots, who are performing some (semi-)autonomous task.

Another direction is to evaluate the performance of a distributed TS compared to a shared TS. Aspects that could be researched are the how communication between TSHs affect performance, and how to can be used to lessen the load on TSCs. Both distributed and shared TS should be evaluated for use in a mobile environment, to see the weaknesses and strengths of both approaches.

Future work for the work on a shared TS presented in this thesis include adding support for multiple TSHs per device, associate TSHs with geolocation, add support for choosing which host to store tuples on, add evaluation of a host when storing tuples, and add support for persistent storage. If each NS also holds information about the approximate geolocation of TSHs, by retrieving the geolocation of its host device, this information can be used by robots to locate TSHs more efficiently. Additionally, it could be interesting to make TSCs gossip about the geolocation of TSHs its been in contact with.

As described in the LIME paper[15], it might sometimes be desired to designate which host to store a tuple at. One example is if a robot has collected data which should be used by one specific device. Making sure the data is stored at that device will ensure that it is always available. If the data is stored at another host, the data might be available at one time, but later become unavailable. Another aspect, related to this, is evaluation of host before storing tuples there. Assuming a robot has collected data in the field, and it is essential that this data is analyzed

as soon as possible. If the robot, on its way back to base, passes over a TSH in the middle of the field, it is likely that this TSH is not connected to other devices. Storing the data here will simply make it unavailable until the host also gets out of the field. It might therefore be a better solution to wait for another TSH to pass the data to.

Lastly, in many cases it might be desired to store the data stored at TSHs in a persistent manner, so not to lose data.

REFERENCES

[1] P. Gomes, "Surgical robotics: Reviewing the past, analysing the present, imagining the future," *Robot. Comput.-Integr. Manuf.*, vol. 27, pp. 261–266, April 2011.

[2] "Predator rq-1/mq-1/mq-9 reaper." `http://www.airforce-technology.com/Projects/predator-uav/`. Accessed: January 26, 2012.

[3] iRobot, "Ground robots by irobot." `http://www.irobot.com/gi/ground/`. Accessed: January 26, 2012.

[4] P. Doherty and P. Rudol, "A uav search and rescue scenario with human body detection and geolocalization," in *Proceedings of the 20th Australian joint conference on Advances in artificial intelligence*, AI'07, (Berlin, Heidelberg), pp. 1–13, Springer-Verlag, 2007.

[5] T. Reed, J. Geis, and S. Dietrich, "Skynet: a 3g-enabled mobile attack drone and stealth botmaster," in *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2011.

[6] A. Foka and P. Trahanias, "Predictive autonomous robot navigation," in *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, vol. 1, pp. 490 – 495 vol.1, 2002.

[7] M. A. Goodrich, T. W. McLain, J. D. Anderson, J. Sun, and J. W. Crandall, "Managing autonomy in robot teams: observations from four experiments," in *Proceedings of the ACM/IEEE international conference on Human-robot interaction*, HRI '07, (New York, NY, USA), pp. 25–32, ACM, 2007.

[8] T. Finin, R. Fritzson, D. McKay, and R. McEntire, "Kqml as an agent communication language," in *Proceedings of the third international conference on Information and knowledge management*, CIKM '94, (New York, NY, USA), pp. 456–463, ACM, 1994.

[9] L. Aszalós and A. Herzig, "A logic for semi-public communication in multi-agent systems," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, AAMAS '02, (New York, NY, USA), pp. 950–951, ACM, 2002.

[10] C. V. Goldman and S. Zilberstein, "Optimizing information exchange in co-operative multi-agent systems," in *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, AAMAS '03, (New York, NY, USA), pp. 137–144, ACM, 2003.

[11] M.-W. Jang, A. Ahmed, and G. Agha, "Efficient agent communication in multi-agent systems," 2004.

[12] N. H. Minsky, Y. M. Minsky, and V. Ungureanu, "Safe tuplespace-based coordination in multi agent systems," 2001.

[13] M. Mamei, F. Zambonelli, and L. Leonardi, "Tuples on the air: A middleware for context-aware computing in dynamic networks," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW '03, (Washington, DC, USA), pp. 342–, IEEE Computer Society, 2003.

[14] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, January 1985.

[15] G. P. Picco, A. L. Murphy, and G.-C. Roman, "Lime: Linda meets mobility," in *Proceedings of the 21st international conference on Software engineering*, ICSE '99, (New York, NY, USA), pp. 368–377, ACM, 1999.

[16] "Javaspaces principles, patterns, and practice." `http://java.sun.com/developer/Books/JavaSpaces/introduction.html`. Accessed: May 9, 2012.

[17] W. Ware, "Linuxtuples, a tuple space server for linux." `http://linuxtuples.sourceforge.net/`. Accessed: January 20, 2012.

[18] "Parrot augmented reality drone." `http://ardrone.parrot.com/parrot-ar-drone/usa/`. Accessed: March 6, 2012.

[19] "Ar.drone sdk and developer guide." `https://projects.ardrone.org/`. Accessed: March 6, 2012.

[20] J. Faigl, T. Krajník, V. Vonásek, and L. Přeučil, "Surveillance Planning with Localization Uncertainty for UAVs," in *3rd Israeli Conference on Robotics,*, (Ariel), pp. –, Ariel University Center, 2010.

[21] T. Krajník, V. Vonásek, D. Fišer, and J. Faigl, "AR-Drone as a Platform for Robotic Research and Education," in *Research and Education in Robotics: EUROBOT 2011*, (Heidelberg), Springer, 2011.

[22] C. Bills, J. Chen, and A. Saxena, "Autonomous mav flight in indoor environments using single image perspective cues.," in *ICRA*, pp. 5776–5783, IEEE, 2011.

[23] K. Higuchi, T. Shimada, and J. Rekimoto, "Flying sports assistant: external visual imagery representation for sports training," in *Proceedings of the 2nd Augmented Human International Conference*, AH '11, (New York, NY, USA), pp. 7:1–7:4, ACM, 2011.

[24] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[25] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *J. ACM*, vol. 31, pp. 560–599, June 1984.

[26] "Python, pickle module." `http://docs.python.org/library/pickle.html`. Accessed: February 29, 2012.

[27] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[28] S. Emami, "Opencv tutorials." `http://www.shervinemami.info/openCV.html`. Accessed: March 7, 2012.

[29] S. D. Levy, "Autopylot." `http://home.wlu.edu/~levys/software/ardrone_autopylot/`. Accessed: March 7, 2012.

[30] B. Venthur, "Ar.drone python api." `https://github.com/venthur/python-ardrone`. Accessed: March 6, 2012.

[31] J. M. Bjørndalen, B. Vinter, and O. Anshus, "Pycsp - communicating sequential processes for python," in *Communicating Process Architectures 2007: WoTUG-30*, vol. 65, pp. 229–248, IOS Press, 2007.

[32] B. Vinter, J. M. Bjørndalen, and O. Anshus, "Pycsp revisited," in *Communicating Process Architectures 2009: WoTUG-32*, vol. 67, pp. 263–276, ISO Press, 2009.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications,"

in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[34] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

APPENDIX A
AR.DRONE 1.0 TECHNICAL SPECIFICATIONS

The following is the technical specifications for the AR.Drone 1.0, taken from the AR.Drone official website[1].

AERONAUTIC STRUCTURE

- High-efficiency propellers
- Carbon-fiber tube structure

MOTORS AND ENERGY

- 4 brushless motors, (35.000 rpm, power: 15 W)
- Lithium polymer battery (3 cells. 11-1 V, 1000 mAh)
- Battery charging time: 90 minutes

FRONT CAMERA

- 93° wide-angle diagonal lens camera, CMOS sensor
- Encoding and live streaming of images over Wi-Fi
- Camera resolution: 640x480 pixels (VGA)
- Video frequency: 15 FPS

[1] http://ardrone.parrot.com/parrot-ar-drone/usa/technologies, (April 13, 2012)

VERTICAL CAMERA

- 64° diagonal lens, CMOS sensor
- Video frequency: 60 FPS
- Allows stabilization even with a light wind
- Resolution: 176x144 pixels (QCIF)

EMBEDDED COMPUTER SYSTEM

- ARM9 RISC 32 Bits 468 MHz
- DDR 128 Mb at 200 MHz
- Wi-Fi b/g
- USB high speed
- Linux OS

SPECIFICATIONS

- Running speed: 5 m/s; 18 km/h
- Weight:
  - 380 gram with outdoor hull (0.8 pounds)
  - 420 gram with indoor hull (0.9 pounds)
- Flying time: about 12 minutes

DIMENSIONS

- With hull: 52.5x51.5 cm (20.7 inches x 20.3 inches)
- Without hull: 45x29 cm (17.7 inches x 11.4 inches)

INERTIAL GUIDANCE SYSTEM WITH MEMS

- 3 axis accelerometer
- 2 axis gyrometer
- 1 axis yaw precision gyrometer

SAFETY SYSTEM

- Expanded Polypropylene hull for indoor flight
- Automatic locking of propellers in the event of contact
- UL2054 battery
- Control interface with emergency button to stop the motors

APPENDIX B
AR.DRONE 2.0 TECHNICAL SPECIFICATIONS

The following is the technical specifications for the AR.Drone 2.0, taken from the AR.Drone official website[1]. The AR.Drone 2.0 is set to be available in June 2012.

VIDEO STREAM

- HD Camera, 720p 30 FPS
- Wide angle lens: 92° diagonal
- H264 encoding base profile
- Low latency streaming
- Video storage on the fly with Wi-Fi directly on the remote device or on a USB key
- JPEG photo
- Traveling, Pan, Crane predefined autopilot modes for video recording

ELECTRONICS

- 1 GHz 32 bit ARM Cortex A8 processor with 800 MHz video DSP TMS320DMC64x
- Linux 2.6.32
- 1 Gbit DDR2 RAM at 200 MHz
- USB 2.0 high speed for extensions

[1] http://ardrone2.parrot.com/ar-drone-2/specifications/, (April 13, 2012)

- Wi-Fi b/g/n
- 3 axis gyroscope 2000°/second precision
- 3 axis accelerometer ±50 mG precision (G is the gravitational constant)
- 3 axis magnetometer 6° precision
- Pressure sensor ±10 Pa precision (80 cm at sea level)
- Ultrasound sensor for ground altitude measurement
- 60 FPS vertical QVGA camera for ground speed measurement

AERONAUTIC STRUCTURE

- Carbon fiber tube structure
- High grade 30% fiber charged nylon plastic parts
- Foam to isolate the inertial center from the engines' vibration
- Expanded Polypropylene hull injected by a sintered metal mold
- Liquid Repellent Nano-Coating on ultrasound sensor
- Weight:
  - 380 gram with outdoor hull (0.8 pounds)
  - 420 gram with indoor hull (0.9 pounds)

MOTORS AND ENERGY

- 4 brushless inrunner motors
- Low noise Nylatron gears for 1/8.75 propeller reductor
- Tempered steel propeller shaft
- Self-lubricating bronze bearing
- Specific high propelled drag
- 8 MIPS AVR CPU per motor controller

- 3 elements 1000 mA/H lithium polymer battery
- Emergency stop controlled by software
- Fully reprogrammable motor controller
- Water resistant motor's electronic controller