NU-Architecture Requirements Document

2013 Final Year Project

June 12, 2013

Status: Draft

Version: 0.6

Todo list

Contents

1 Document Notes

Version	Changes	Author
0.1	Initial Template	Jake Woods
0.2	Non-Goals, Requirements (Multithreading, Portability, Performance, Consistent Interfaces, New Components), Extension Goals (ROS Support)	Jake Woods
0.3	Initial Latex Version, Requirements (Networking, Debugging, Unit Testing)	Trent Houliston
0.4	Current and Proposed System Diagrams, Placeholders for sections	Trent Houliston, Jake Woods
0.5	Updated Requirements Diagrams and Descriptions (Current and Proposed)	Jake Woods
0.6	Updates and added more diagrams and massively extended the overview.	Jake Woods

2 Scenarios

This is a high level overview of how the architecture will be used by the client. In our case the client is a programming team so we need to provide details of how the API will be used to implement existing features. We should also provide some examples of how new features would be added. Because this document is targeted at a programming team we should include some code examples. However we shouldn't clutter all the pages with piles of code so please try and keep the examples simple!

Right now I'm arguing that scenarios and non-goals come first. This is so we can set the scene in the readers mind and show them the problems that need to be solved. Scenarios is also a good place to talk about how "here's how this would have worked in the old system. As you can see our system is a million times better because instead of being (impossible/a huge pain in the ass/computationally infeasible) it only took 15 minutes of work!

Example Scenarios: Field Changes, Number of Players Change compare old/new system.

2.1 New Component Added to System

Write the scenario for a new component added

2.2 Debugging When Bug is Found

Write the scenario for a bug occurring

2.3 The System is moved to a new platform

Write the scenario for the system is moved to a new platform

2.4 The Robot is used to perform a new task

Write the scenario for when the robot is given a new task (e.g. catering for university)

3 Non-Goals

The architecture will not be making any modifications to the existing code base beyond the changes necessary to integrate the existing code into the new architecture. This project isn't concerned about the specific algorithms used or the internal behaviour of the robot. We are also not concerned with improving the physical architecture of the robot such as improved motors or cameras.

For example the following goals are non-goals: Changing any of the algorithms currently used in the system. Such as changing the vision system from *current algorithm* to *some other algorithm* Improving the physical architecture of the robot (adding new cameras, better motors, etc...)

Comment: We need to get non-goals out of the way as soon as possible. These are things like rewriting modules, improving algorithms and other things that are out of scope for the project. It's important that we clarify these so clients don't go through this document thinking "why haven't they addressed the *insert non-goal here* issue?"

3.1 Algorithm Changes

Write the non-goal that we will not do changes to the algorithm

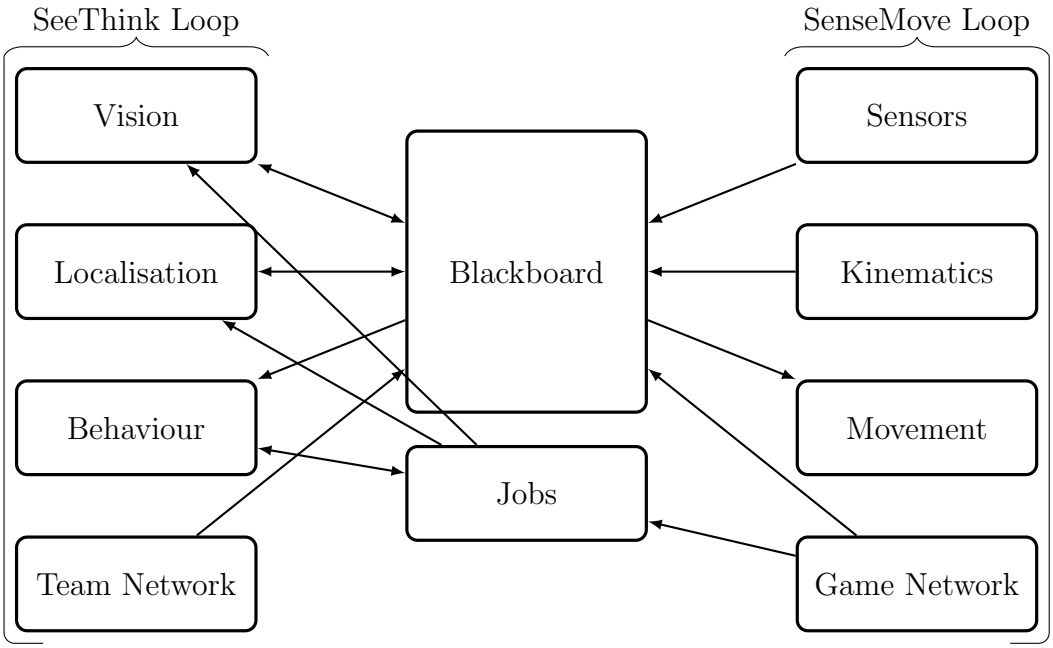

Figure 1: The existing architecture

3.2 Hardware Changes

write the non-goal that states that the project will not cause hardware changes

4 Overview

Here's where we provide a high level overview of the proposed system, its major benefits and also any relevant system diagrams. This is also a good place to give an overview of the old architecture and how the new system compares from a high level point of view.

4.1 Existing Architecture

Provide a high level overview of the old system and the way it works

The existing architecture is composed of a number of interconnected components that communicate through individually defined interfaces and mechanisms. For example: The vision system communicates utilising a two step process. First the vision system requests that the sensors retrieve a new frame. It then waits on the sensor system to place the frame information on blackboard. Once the frame information is placed on blackboard the vision system then reads the information from blackboard and continues

it's processing. This is only one of the myriad of ways in which two components can communicate and we believe that this reduces the ability to both learn and modify the system.

Expand the description of the Current Architecture

4.2 Proposed Architecture

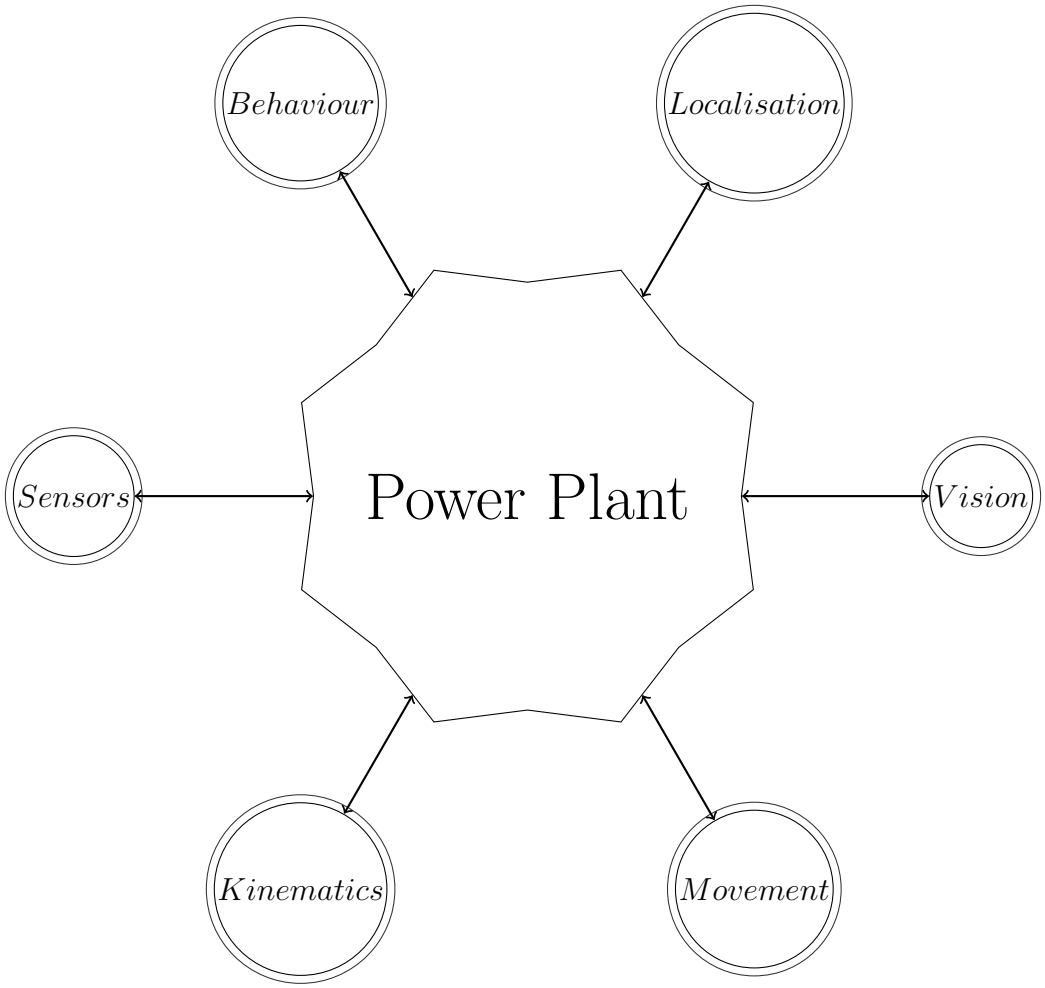

Figure 2: The proposed architecture

The central authority of the proposed architecture is known as the "Power Plant" which is responsible for coordinating the various functions of the architecture and works behind the scenes to provide message handling. Only one power plant exists can per program and the majority of the time the user won't even need to think about it.

The point of interaction that most users will see is called a "Reactor". When a user wants to create a new feature they create a new class that inherits from Reactor. This

gives them access to the two key functions of the proposed architecture: **On** and **Emit**. *On* allows users to specify reactions which can be though of as callbacks that are called when new data comes in. For example: A reaction that is called whenever new image data is retrieved. *Emit* allows reactors to send out a new piece of data such as the sensor reactor using *Emit* to send new image data every time it reads the camera hardware. Reactors also provide one special type of event named **Every** which lets you subscribe to an event that is guaranteed to tick at consistent intervals. See Figure 3 on page 14 for an example of how *On*, *Emit* and *Every* are used.

One of the key advantages of this design is that components are loosely coupled and only depend on the data format not changing. This means that you could replace the hardware-dependant camera system with a completely different module that reads a pre recorded video stream for testing purposes. It also means that adding a new component to the system merely requires that you know what sort of data you want to access. Gone are they days of frantically trying to track down the obscure method used to access the kinematics system. Instead you simply tell the system that you want the kinematics data and then you have it!

Unlike humans, computers are becoming exceptionally good at multitasking. Unfortunately the current system only takes advantage of at most two cores. You wouldn't ride horse that only used one of it's legs so it's equally important that the proposed architecture takes full advantage of the robots ever-improving multitasking capabilities. The proposed architecture takes care of multithreading transparently so the NUbots team can focus on figuring out the important questions such as how to get the robot to deliver pizza to the lab. When a reaction is triggered it doesn't necessarily run immediately. Instead the reaction is put into a blocking priority queue and then executed on a worker thread owned by the power plant. For an example of how this system works see Figure 4 on page 15.

The client has also expressed how important it is that the robots be able to easily communicate. Networking is a difficult and error-prone process which is why it's imperative that the architecture provide a simple mechanism for robots to communicate. The proposed architecture allows you to treat reactors on other robots as potential targets for your data. This means you can emit data on one robot and receive it on any of the other robots. See Figure 5 on page 15 for an example of how the networking system allows robots to cooperate and share data.

Expand the description of the Proposed Architecture

Explain the diagram that all the components are loosely coupled

5 Requirements

This is the nitty-gritty section of the document. You can expect this section to be many times longer than any other section. Here we need to list every single requirement: What it is, why it's important, how we can test if we've achieved it and any important details about the requirement.

One of the most important things to include here are any decisions and assumptions that have been made. For example:

5.1 Multiprocessing

The architecture must take advantage of all CPU cores

The existing system is currently only able to take advantage of two cores, and one of those cores spends a lot of time doing very little work. Unfortunately writing multi-threaded code is difficult and distracts programmers from more important things such as improving the speed at which a robot can backflip by 5%. To account for this the proposed API must provide a way for programmers to easily utilise the full CPU power of any robot platform.

This requirement is going to become increasingly important as time goes on. The current trend in computing performance is to add more cores and in the cutthroat world of robotic soccer it is of paramount importance that we utilise all of our resources.

From an API point of view the ideal acceptance test for this requirement is to determine how often a programmer needs to think about multithreading at all. We could analyse the code to determine what sections need to include multithreaded primitives and from that percentage determine how many times the API failed to provide the proper multithreaded abstractions.

From a performance and hardware point of view we can measure CPU utilisation on various platforms and compare it to the old system.

Technical Note: We're assuming that we aren't going to be using many single-core machines. The API should still support single threaded machine but from a performance point of view we're assuming that additional cores is the way to go.

5.2 Robot Platforms

The architecture must be portable to new platforms

The existing system has a lot of complicated logic that is used to support a few different platforms. It's currently not possible to easily swap out the Darwin motor components for the AIBO motor component due to the tight coupling of systems.

The new architecture should provide a way to easily slot in platform dependant components. For example it should be possible to remove the hardware-dependant Darwin camera component and replace it with an AIBO component with minimal to no modification of other components.

Portability can be measured by determining the amount of code that depends on specific hardware or platforms. For this exercise Unit Tests can be considered another platform so we could determine the portability by replacing hardware-dependant components with mock components.

Technical Note: Obviously if the format of the data changes the systems that rely on that data need to change as well. We're looking to reduce unnecessary changes due to API bloat.

Double check that AIBO is the appropriate name for the platform.

5.3 Performance

The architecture must have acceptable performance

Robotic platforms have very strict requirements about how often motors need to be sent commands. If these performance requirements aren't met the robots are approximately as useful as a very ambitious block of wood. Because of these requirements the proposed architecture cannot impose large processing or resource costs to the existing system.

A good example of a system that imposes heavy performance costs is ROS (Robot Operating System). ROS has made a number of trade-offs to facilitate distributed multi-language multi-platform systems but those trade-offs have resulted in unacceptable performance implications for smaller robots such as the Darwin.

The proposed architecture should be optimised to run efficiently so it doesn't take valuable resources away from critical computations. Ideally the architecture should be structured in such a way that it can assist the NUBots team in writing efficient code. The first key indicator of performance is to determine the ratio of time used by the architecture vs. the time taken by actual components. Ideally the ratio should be so small as to be practically insignificant. A good architecture should also provide the tools to measure performance.

We can also measure the speed of the old architecture vs. the new architecture to determine what improvements have been made.

5.4 Component Interfaces

The architecture must promote consistent interfaces between components

In the existing system there are a number of ways components can communicate all of which have their own conventions, quirks and pitfalls. The biggest problem with this is it greatly increases the complexity of the system and also causes increased coupling due to all the different ways components can communicate.

The proposed system needs to provide a unified mechanism for components to communicate. However this unified mechanism shouldn't force us to remove any existing functionality and as such must be able to accommodate or replace any of the existing communication styles. By providing a unified mechanism we can greatly reduce the complexity of the system.

This requirement can be measured by analysing the mechanisms that components use to communicate. For example we can look at all the places the Camera system communicates with the Vision system and determine the number of unique ways in which they communicate. Additionally we can measure the suitability of the unified mechanism by ensuring that it doesn't force us to remove any existing functionality.

5.5 Adding Components

The architecture must make it easy to add new components to the system

Adding a new component to the existing system is currently a highly perilous journey of discovery involving knowledge of huge portions of the existing system to achieve and copious amounts of prayer. Because the existing communication graph is basically a giant yarn ball held together by faith every new component exponentially adds more complexity to the existing system and makes the next new component even harder to add.

This is one of the key components the new architecture needs to solve. Ideally if you want to add a new component you should only need to know about the inputs and outputs of your component. The proposed architecture must provide a mechanism to simplify adding new components to the system.

This requirement is best measured by comparing it against the old system. Take some hypothetical component and look at how many systems you would need to touch to add it. For example we could consider adding a new behaviour system to handle catering at university functions. We could then analyse how many components would need to be understood and changed to add this new components we could also develop a metric for analysing the extent of the changes.

Consider defining analysis metric in this document

5.6 Networking

The architecture must provide methods to easily perform network communication

Communication is a vital part of any team activity, without communication there is no way to work on team behaviour, or share useful information. The current system does have a networking solution. However it is difficult to use and this deters coders from using it to better the team abilities of the robots. By providing a simple and intuitive interface to send and receive networked data, a greater range of possibilities of team behaviour and distributed computing become accessible to the NUBots team.

The new architecture must be able to automatically find and communicate with any robots that are currently on the network without configuration (auto discovery). It also must be able to serialise packets of data into a binary format for the majority of cases, and allow the user to provide a serializer for any edge cases. The cases that should be covered by the system as a minimum are any data type that contains only Plain Old Data (a container of basic data types only), or an instance of a Protocol Buffer.

Provide a citation for protocol buffers

This requirement is measured by the amount of code that is required to both send and receive network packets from other robots, as well as how efficient the binary representation over the network is. For example, take the case of the two robots bragging about how they went after the game. Being robots, it would be inefficient to communicate using a natural language such as English they would not be able to convey their level of awesome quickly enough (running out of bandwidth), as such all information exchanged should be in a binary format. Also, as the robot is made up of many individual components, the communication between these components, must use the same communication channel, but have different origin and destination points

5.7 Debugging Tools

The architecture must provide tools for debugging components

Debugging in the current system is a very difficult and painful process. Since all of the components are so tightly integrated with each other, this means that if an error occurs in one system, it is difficult to locate where the source of the error was. For example, in the current system an issue exists where if the camera is not connected, the vision system (the next system in the pipeline) will progress to the point of classifying the image before the robot crashes. This makes it appear that the bug is located in the vision system, however it is actually located in the camera reading system.

The new architecture should provide tools within it that allow the debugging of both errors with the systems, as well as the performance between the systems. Using the tree style model of communication between components, it is possible to output this tree into

a format that is understandable. This would allow the person debugging the system to see where each data packet that was used in the system of interest came from, and if enabled the contents of those data packets should also be available in order to replay the scenario that caused the error.

Provide a measurable for the debugging tools

5.8 Testing

The architecture must provide tools for unit testing individual components

Unit testing is a very important concept as it allows a number of tests to be performed on individual components in the system. This can help to identify and kill bugs before they even leave the development environment. By enforcing greater isolation between the components, the architecture is able to make it much easier to test an individual component by sending it fake data and validating the results.

The system must provide a testing harness that makes it easy to test in isolation a component of the system. This testing harness should be able to wrap around any of the individual components that make up the system and, without modification to the component itself, send fake input to the component and capture and validate any output from that system. This should allow the system to be unit tested thoroughly such that the number of bugs that exist when the code is executed on the robot is much lower.

This requirement is measured by comparing the difficulty of testing a component between the current system and the new system. This includes both the difficulty of extracting the component from the system to be tested without other components (isolating the component) as well as the difficulty of testing its API and results once it has been extracted from the system.

6 Extension Goals

This section will outline extra goals that are not requirements but would be useful to have if we have extra time. None of these should be implemented before a Requirement unless the requirement implementation trivially adds an extension.

6.1 The architecture could support ROS (Robot Operating System) integration

There are a number of components that could be useful from the existing ROS codebase. The proposed architecture could provide systems to make ROS component integration easy. Currently the existing system doesn't contain any facilities for ROS integration.

Additionally no other RoboCup teams have any degree of ROS support. Having ROS support would give us a crippling advantage over the other teams.

This goal can be measured by checking if we can integrate a ROS component.

Improve/extend. Validate the claim about other Robocup teams

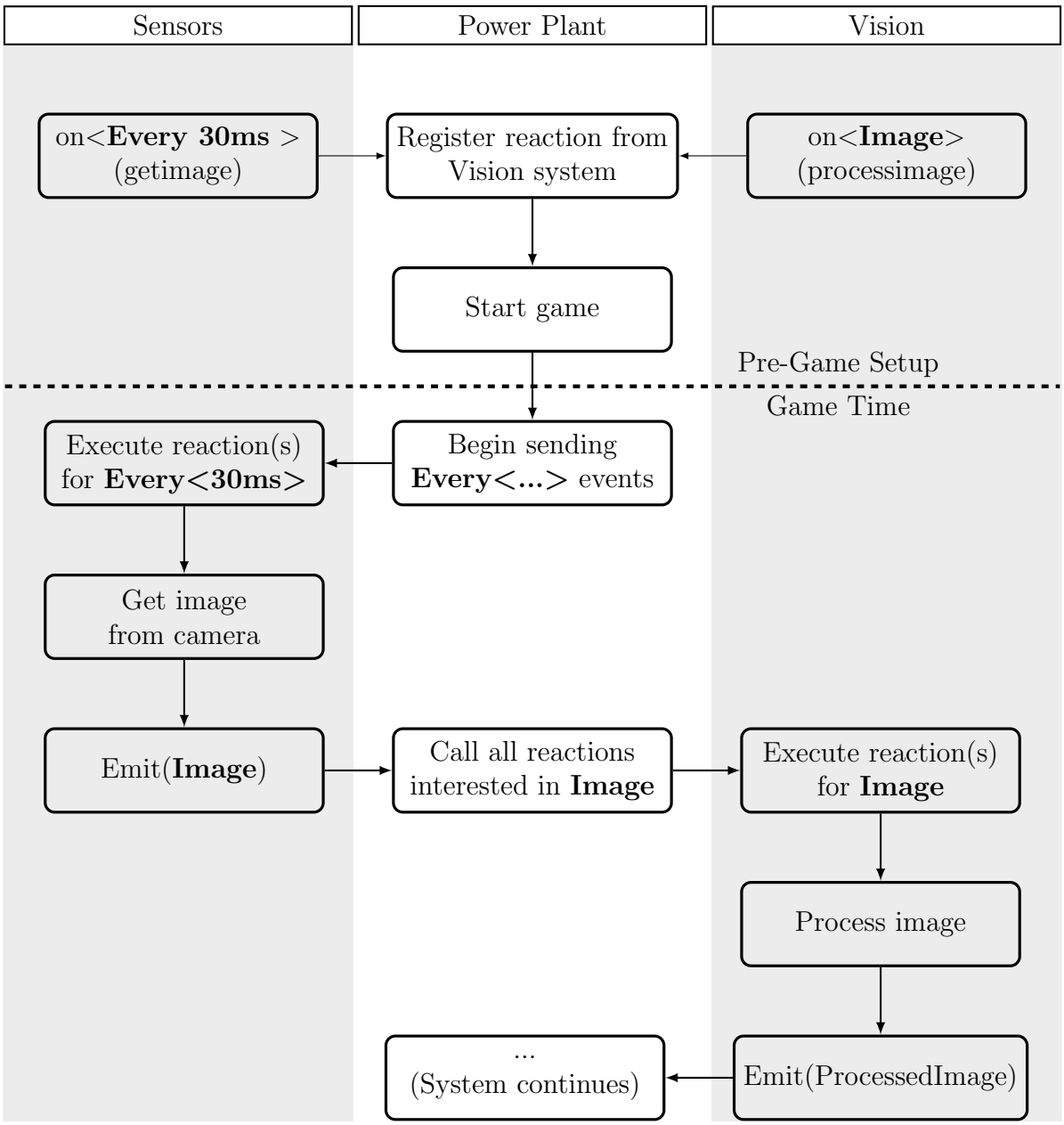

Figure 3: A flowchart example of how **On**, **Emit** and **Every** work

Figure 4: An overview of the Power Plant threading system

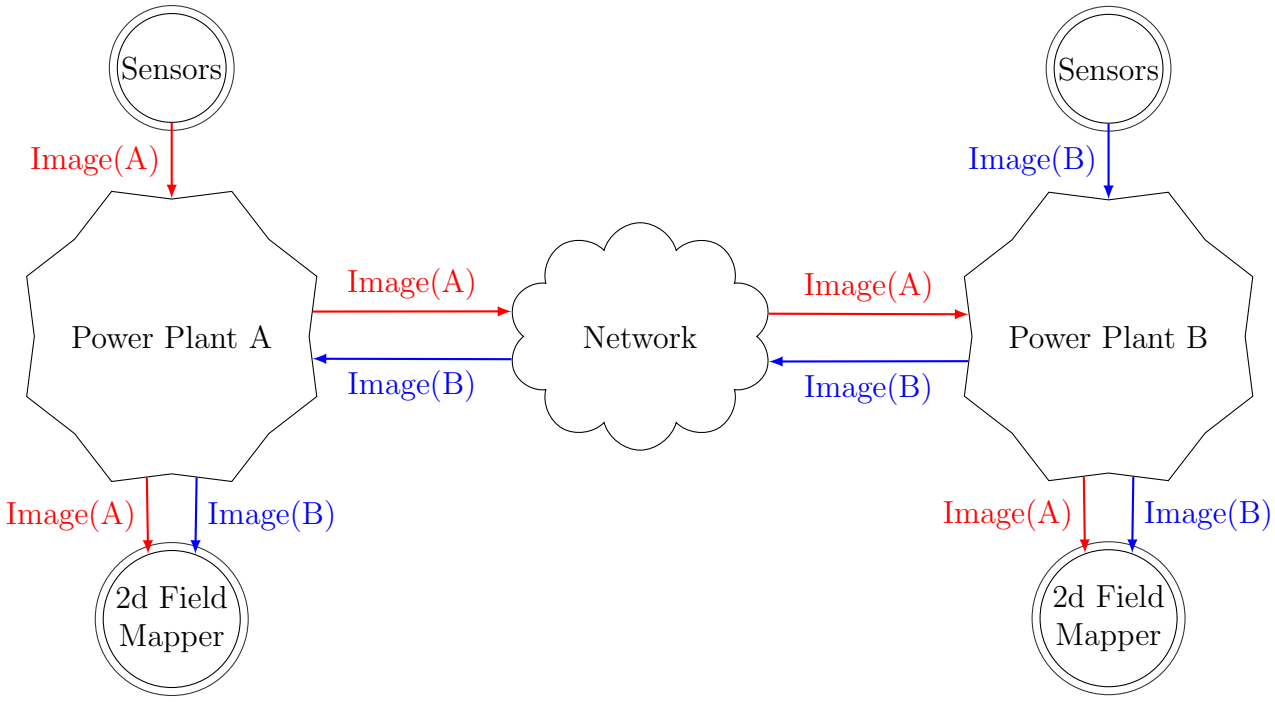

Figure 5: An example of how robots can communicate camera data over a network to build a 2d map of the field