RECOVERY METHODS FOR CLIENT-SERVICE BASED ACTION ENTROPY ACTIVE SENSING

By: ALEXIS SCOTT

Submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science and Data Science

CASE WESTERN RESERVE UNIVERSITY

August, 2023

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Alexis Scott

candidate for the degree of Master of Science

Committee Chair

Vincenzo Liberatore

Committee Member

Vincenzo Liberatore

Committee Member

M. Cenk Cavusoglu

Committee Member

Orhan Ozguner

Date of Defense

June 1, 2023

*We also certify that written approval has been obtained

for any proprietary material contained therein

Table of Contents

List of Tables

List of Figures

Recovery Methods for Client-Service Based Action Entropy Active Sensing

Abstract

By: ALEXIS SCOTT

Action entropy active sensing represents huge leaps forward in the field of active sensing. Previous papers have found that while action entropy increases the accuracy of the simulation, it also increases the time required to calculate sensing actions. The time cost can be reduced by implementing parallel processing that splits the computations over multiple cores. This can be done on a local machine and via implementation of the cloud. This paper focuses on implementing a detection and recovery mechanism for a potential cloud crash. The communication is facilitated through Docker and OpenVPN, using the ROS client-service architecture. This paper adjusts the methodology of the existing model to make it more robust to failures. The simulation results indicate that the time to recovery is feasible, and that this new structure increases the reliability of the existing model.

Chapter 1 Introduction

As the complexity of tasks assigned to robots increases, the computation power needed for these tasks increases as well. Although currently most tasks can be completed via a local processor, the need for cloud technology will increase. However, cloud technology requires network communication and is susceptible to crashes. In some cases, a delay or pause in computation is acceptable. In other time sensitive situations however, it is not. Consider the case where a robot is performing surgery, and deciding which action to take while suturing a wound. A crash that leads to a long delay, or pauses the program until the cloud component is fixed entirely would be unacceptable in this situation.

In these cases it is important to have a reliable recovery or backup mechanism. This project focuses on a previously implemented action entropy sensing action calculation through a client-service architecture. The existing architecture did not account for the need for recovery. This project adjusts the structure so that a recovery mechanism is possible. One is implemented, the time to recovery is measured, and the time costs of running the simulation entirely on a local machine versus a cloud and local machine are discussed and compared.

The rest of the paper will proceed as follows, Chapter Two: Background, Chapter Three: Related Work, Chapter Four: Architecture, Chapter Five: Methodology, Chapter Six: Simulation and Results, Chapter Seven: Discussion, Chapter Eight: Conclusion and Future Work.

Chapter 2 Background

This chapter will cover the necessary background to understand the project. It will discuss the theory of cloud robotics, the robotic operating systems or ROS used in the project, and the various other softwares implemented in the project.

2.1 Cloud Robotics

As computing needs increase, the motivation to offload processes and share data among systems increases as well. For larger tasks more storage space and computing capacity is needed, but it is often expensive to upgrade the hardware of existing systems. The cloud provides a valuable solution to this problem. The cloud allows systems to access a shared pool of information, without the hardware to host it independently [1]. Additionally, the cloud can perform virtual computing to offload heavy computation tasks from local computers [2]. As the cloud has become more reliable, the use cases for it have expanded into robotics [2].

Cloud Robotics encompasses a wide variety of different programs. Broadly, it can be defined as any robot or automation system that uses code or data from a network to support the operation [3]. This allows for the system to have a local component to handle specific elements of the task, while still following under the broad definition of cloud robotics. There are four main benefits to using cloud robotics, which are Big Data, Cloud Computing, Collective Robot learning, and Human computation [3]. This project primarily focuses on the Cloud Computing aspect, utilizing the cloud to get more resources for parallel grid computing.

Since different robotic systems have different communication and computation needs, cloud robotics does not have a one size fits all solution [1]. Therefore a programmable and adaptable solution is needed on the cloud. Many companies have proposed different solutions over the years. Some solutions involve specific software and hardware structures for the involved components, while others are more adaptable to different systems.

Some of the most notable solutions include AWS Robomaker, which gives programmers the ability to develop code in the cloud, test it on the gazebo simulator and then apply it to their own machines [1]. Another proposed program is ABC Robot. This service focuses on vision based perception, objection recognition, facial recognition, and text recognition [1]. While the software it provides is useful, it does require robots to be compatible with its specific system.

More general cloud parallel computing can be found from many commercial sources, like Google's Compute Engine, Amazon's Elastic Cloud Computer, and Microsoft's Azure [3]. While these computing resources are not explicitly designed for use with cloud robotics, the additional computing capabilities they provide are extremely useful in robotic planning problems [4].

This project does not use any pre-existing robotic software designed to function with the cloud. Instead, it uses OpenVPN and docker containers to simulate a local network, in combination with the Robotic Operating System or ROS which facilitates the robotics side of the system. However, it still utilizes the resources provided by the cloud and builds on theories proposed by previous cloud robotics research.

2.2 Robotic Operating System (ROS)

The Robot Operating System or ROS is an open source library for robotics work. It allows researchers to develop code that can be shared among systems with a common framework [5]. ROS is a language neutral messaging system, so it works well for many projects and is easily adaptable [5].

ROS programs operate through nodes, which are analogous to software modules. Most systems end up being composed of many different nodes due to the modular nature of ROS [5]. In order to communicate with other nodes, nodes pass messages to each other. Messages can consist of primitive data types, or be customized depending on the needs of the system. Messages are published to topics, which are defined with a string name [6].

This project used the client-service architecture. A ROS service is a callable node with a strictly typed request, and response structure [6]. This structure is then called by a client node, which formats the request messages to fit the service's needs and processes the response from the service. This type of communication works better for synchronous messaging than the other option, publish and subscribe.

The communication type used for this layer of transport is TCPROS [7]. It uses standard TCP/IP sockets to transport data between nodes. Different fields route to service nodes or publisher nodes. A service is always required to reply with an 'ok' byte in response to each service request message. This allows for detection of when a service call fails [7].

2.3 ROS Master

Due to the peer to peer nature of the ROS architecture, an additional component is required so that nodes can find each other [5]. The ROS master is a node that must be run at start-up for all other nodes to communicate with each other. The master node tracks services and other nodes, and allows them to pass messages to each other [8]. Once the nodes have located each other through the master they are able to communicate on a peer to peer connection, however the master must be running for the eternity of the system's operation [8]. If ROS master crashes, the nodes are unable to communicate with each other, or run their own functions.

2.4 Docker

For ease of portability and network simulation this project was run on Docker containers. Docker containers are a means of encapsulating code and all of its dependencies into an isolated environment that can be run with simple commands [9]. The environments are created as images, and then turn into containers when they are run [9]. They are ideal for cloud environments as they can be easily deployed on a number of different machines, with limited setup. The Docker architecture is expanded on in section five.

2.5 OpenVPN

Openvpn was used for communication in this project [10]. It is a virtual private network or VPN. Openvpn uses an encrypted tunnel to send packets over a network [10]. This allows traffic to pass securely and privately even without a local connection. This allows the program to treat network traffic as though it is on a local network. Since they are connected as though through a local network, ROS is able to send information through the tunnel and communicate with the other nodes on the network. This adds an additional layer of security that ROS requires, as ROS is unable to pass messages through non-local networks.

2.6 Amazon Web Services (AWS)

Amazon Web Services is a cloud computing platform that provides web services to both companies and individuals [11]. Amazon Web Services was used for this project, as they provide reliable and scalable virtual machines for the user [11]. Specifically, an elastic cloud compute or EC2 instance was used for this project.

The EC2 instance is a scalable virtual machine that comes preset up with various memory allocations and operating systems [12]. This project used an Ubuntu c5.4xlarge EC2 instance with 16 virtual cpus, and 8 GBS of storage. This AWS instance is referred throughout the project as the cloud, and is used to simulate an environment where work needs to be offloaded from a local machine for faster processing.

Chapter 3 Related Work

3.1 Task Action Entropy

Most robotics problems are continuous in nature and thus pose a unique challenge in sequential decision making [13]. Various state space planners have been developed in an attempt to address the problem of sequential decision making. A sensing action is an action that has no effect on the environment, but provides the state space planner with useful information about it [13]. Active sensing is the act of choosing the best sensing action. [13]. The sensing actions are chosen in order to reduce state uncertainty, and so the "best" sensing action is one that reduces the state uncertainty the most [13].

The action-entropy active sensing method proposed by Greigarn, et al uses an active-sensing method in combination with a state space planner in order to solve continuous state problems. It uses particle filters to model belief propagation, mapping the belief over the state space based on the task planner's provided policy [13]. A nearest neighbor method gets the entropy of the task action, and then a sensing action is chosen to minimize that entropy [13].

The action space can be divided into two spaces when the sensing action does not affect the overall state, an action space that only affects sensing called the sensing action space, and a space that changes the state of the system called the task action space [13].

The state transition model used is a probability density function of the next state given that a specific task action is performed at a state. The measurement model is a probability density function of the measurement given that the sensing action is performed at that same state [13]. The belief represents the information that the robot has about the state of the environment.

The following equation is used for belief:

$$b(x_t) = p(x_t | b_0, u_{1:t}, v_{1:t}, z_{1:t}), \forall x_t \in X. (1)$$

Where $b(x_0) = p(x_0)$ is the initial belief. t is dropped where it will not cause confusion for the rest of the calculations [12]. Predicted belief is calculated with the following equation,

$$\bar{b}(x') = \int_x p(x' | u, x) b(x) dx . (2)$$

And then updated with

$$b(x) = \eta p(z | v, x) \bar{b}(x). (3)$$

This work also uses a state space planner for task related planning, referred to as the task planner [13]. The planner is divided into submodules as shown below

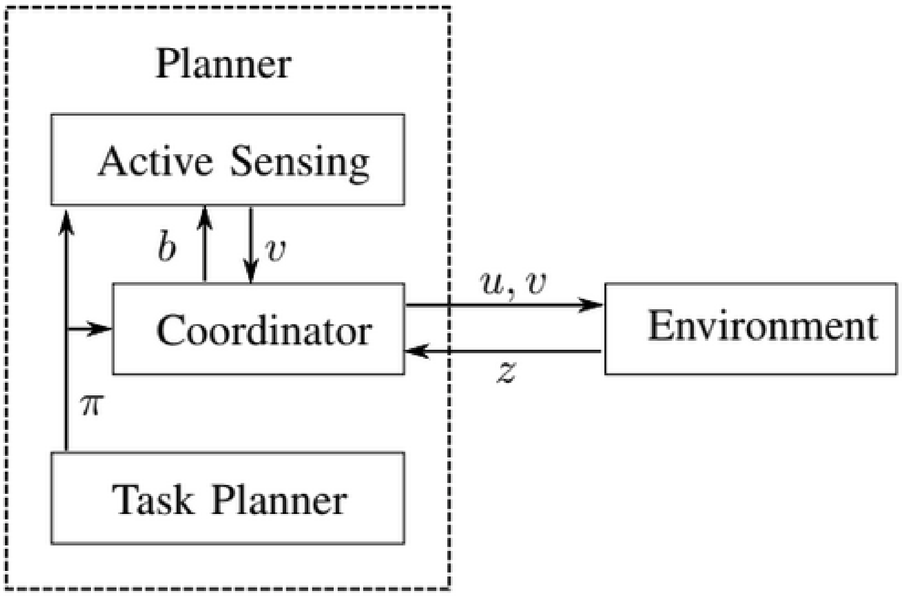


Figure 1. Planner Structure

The full process of the system is as follows [13]:

1. At initialization the planner will solve the planning problem in the state space for a policy and then pass the policy to the active-sensing module. The coordinator will pass the internal belief to the active-sensing module.
2. The belief is then mapped through the policy by the active-sensing module to calculate task action entropy and return the sensing action that minimizes the task action entropy.
3. The coordinator then performs the sensing action and gets a measurement, updating the belief according to the measurement.
4. Then the coordinator selects a task action based on the most likely state.
5. The task action is performed and the coordinator updates its internal belief.

The sensing action is the minimizer of the entropy as follows

$$v = argmin\, h(U|Z;\ v,\ \bar{b})\ where\ v \in \mathrm{V}. (4)$$

Action entropy selection was found to have higher average rewards than state entropy, as well as taking less time [13]. This made it a viable candidate for task selection that is expanded on in future works as well as this project.

3.2 Cloud Task Action Entropy

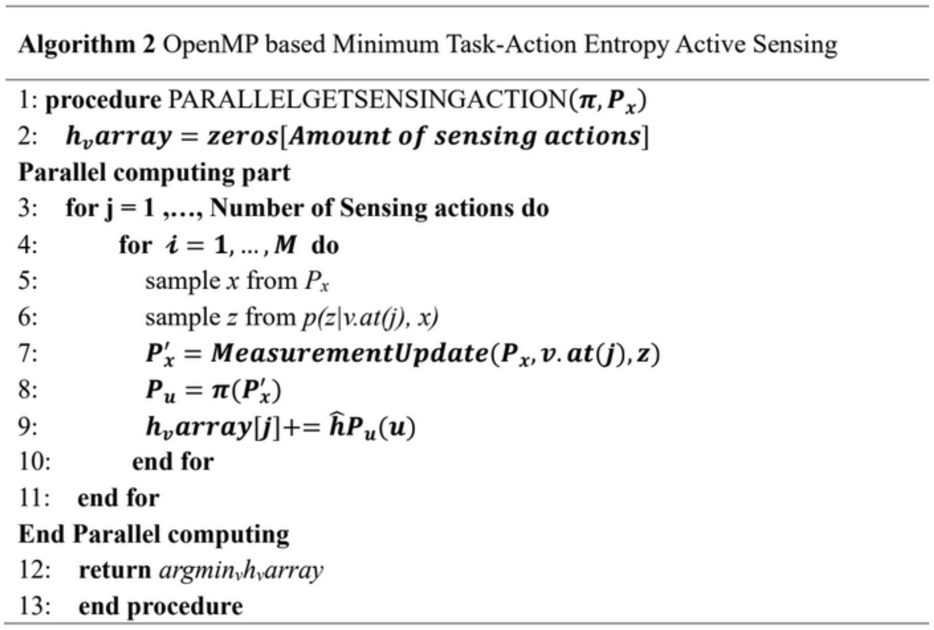
The action entropy approach to state space planning represented a leap forward in the accuracy and efficiency of the state space planning solutions. However, improvements still existed. Liu proposes a model that adjusts the computation of the various actions so that it can take place in parallel [14].

The first step in this new model is to divide Greigan's planner into two parts [14]. The local planner maintains only the coordinator, while the cloud planner has a coordinator, an active sensing module, and a task planner. The cloud portion is also given a virtual environment with which to simulate task action and update the belief [14].

In this setup the local sends a request to the cloud, the task planner calculates the sensing action for the state space and returns it to the local [113. The local then calculates and sends the updated belief which is mapped to calculate the best task action which is returned. The coordinator on the local side then performs the task action and updates its belief of the state space. This cycle continues until either the amount of steps are exceeded or the goal is reached.

A publish and subscribe method of communication between these two portions is discussed in the paper, but ultimately deemed ineffective. It had an overly high communication time for each step of the process, and overall was not a good fit [14]. The client and service model however showed promising results and is the model expanded on in this project [14].

The initial algorithm contained two for loops, one inside of the other, where the outer loop was the for each loop and the inner loop did the Monte Carlo simulation [14]. If split up and computed at the same time, the order of the simulation may be random, which leads to unacceptable results. To apply parallel computing without this error, the algorithm is modified to store the sum of the cumulative entropy in a position in an array where the position corresponds to each sensing action [14]. After the parallel computing is complete, the lowest cumulative entropy is found from the array. The following method is used for this process.



Algorithm 2 OpenMP based Minimum Task-Action Entropy Active Sensing

```
1: procedure PARALLELGETSENSINGACTION($\boldsymbol{\pi}, \boldsymbol{P_x}$)
2:   $\boldsymbol{h_v array = zeros}[\boldsymbol{Amount\ of\ sensing\ actions}]$
Parallel computing part
3:   for j = 1 ,…, Number of Sensing actions do
4:       for $\boldsymbol{i = 1, \ldots, M}$ do
5:           sample $x$ from $P_x$
6:           sample $z$ from $p(z|v.at(j), x)$
7:           $\boldsymbol{P'_x = MeasurementUpdate(P_x, v.at(j), z)}$
8:           $\boldsymbol{P_u = \pi(P'_x)}$
9:           $\boldsymbol{h_v array}[\boldsymbol{j}] \mathrel{+}= \boldsymbol{\hat{h} P_u(u)}$
10:      end for
11:  end for
End Parallel computing
12:  return $argmin_v h_v array$
13:  end procedure
```

Figure 2. Task Action Entropy Algorithm

The revised model was found to perform faster than the original model. Applying multiple cores to the processing also improved the model [14]. The local version was found to perform slightly better than cloud and local version when the appropriate amount of cores were available.

Chapter 4 Architecture

4.1 The Docker Structure

This project uses Docker to manage each of the respective processes. Docker allows for the processes to be packaged up and deployed with ease, without having to install the robotic operating system on each individual machine.

The Local Machine has a docker container with both the client and the backup service built. This container also runs the ROS master and an OpenVPN client. When the container is started ROS master, the ROS client, and the OpenVPN client are running. The backup service only starts if triggered by the recovery mechanism.

The Cloud Machine has a docker container with the regular service inside of it. It uses the network host flag to connect to the OpenVPN server running on the AWS instance instead of hosting the service within the container. On start up it runs the cloud service for standard operation. Both containers can be easily built and run with a single command, with only Docker installed on the machine. This allows the processes to run on machines with smaller memory as well. This setup was the one ultimately used in the project.

4.2 Kubernetes Structure

Kubernetes was explored as an option for this project, but ultimately was not implemented. It was considered for container management. It would have provided an additional way to scale the container size and experiment with that aspect of the project. However, the primary benefit of Kubernetes is its ability to manage multiple containers that need to stay up consistently.

For this project, only two containers are implemented and they are on different systems and need to be managed separately. This would mean having a separate Kubernetes cluster for each of the Docker containers. While this was possible, it added a layer of complexity that was not needed to the project. Ultimately the project did not focus on scaling container size either, and so Kubernetes was not implemented.

4.3 Great Bear Structure

Another option that was explored but ultimately not used in the project was Cisco's Great Bear cloud management system. Similar to Kubernetes, Great Bear would have served as a way to manage the various containers and keep track of their status. It used Docker and Kubernetes as well as its own management system.

A few problems came up while trying to implement Great Bear that ultimately meant it was not feasible for the project in its current form. Great Bear was primarily designed to track programs on a high level, making sure that they were up and running. This project required a level of fine tune control that meant it still needed to be monitored via Kubernetes even while using Great Bear.

Additionally, the network requirements of ROS meant that a VPN was required. Getting a container to run both the VPN and Great Bear ended up not being possible. The conflicting network requirements created problems that were not feasible to fix in the given timeframe. The overhead of setting up Great Bear every time a container needed to be reloaded, also meant that Great Bear was not a good fit for this instance of the project.

Chapter 5 Methodology

5.1 Hardware Specifications

The cloud machine in this project is an AWS Ubuntu c5.4xlarge EC2 instance with 16 virtual cpus, and 8 GBS of storage. The previous project used an Ubuntu instance with 36 virtual cpus, but since only 10 cores were required for the optimal speedup this instance was scaled down. The local desktop that was used had an Intel Core i7-8700 CPU with 12 cores and 16 GB of memory. It is running Ubuntu 22.04, with the project running in a Docker container.

Though the da Vinci robot was not ultimately used in this version of the project, a hardware diagram with how the cloud and local computer in this project would communicate with it in standard operation is included below for reference.

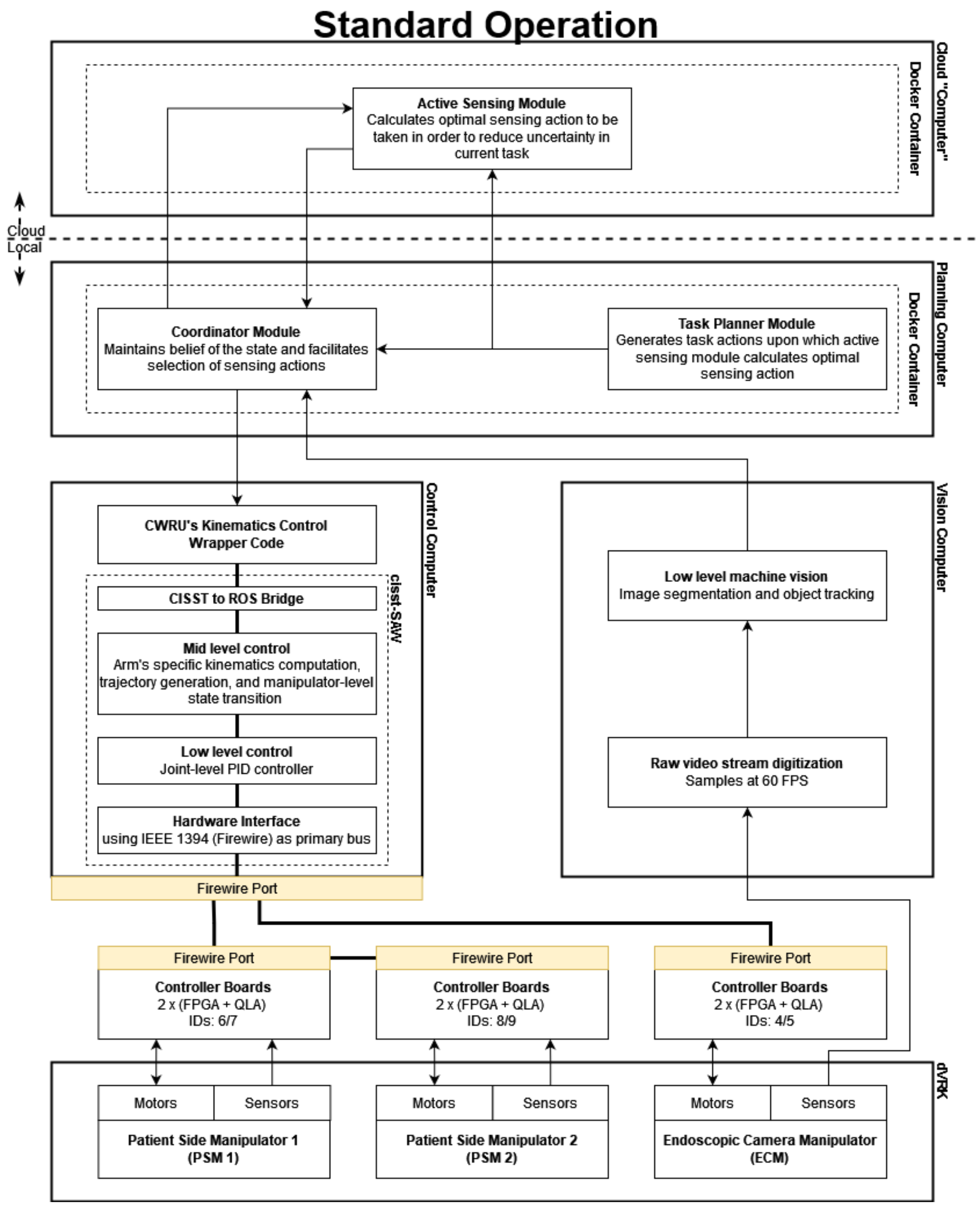

Standard Operation
Cloud "Computer"
Docker Container
Active Sensing Module
Calculates optimal sensing action to be taken in order to reduce uncertainty in current task
Cloud
Local
Planning Computer
Docker Container
Coordinator Module
Maintains belief of the state and facilitates selection of sensing actions
Task Planner Module
Generates task actions upon which active sensing module calculates optimal sensing action
Control Computer
CWRU's Kinematics Control Wrapper Code
cisst-SAW
CISST to ROS Bridge
Mid level control
Arm's specific kinematics computation, trajectory generation, and manipulator-level state transition
Low level control
Joint-level PID controller
Hardware Interface
using IEEE 1394 (Firewire) as primary bus
Firewire Port
Vision Computer
Low level machine vision
Image segmentation and object tracking
Raw video stream digitization
Samples at 60 FPS
Firewire Port
Controller Boards
2 x (FPGA + QLA)
IDs: 6/7
Firewire Port
Controller Boards
2 x (FPGA + QLA)
IDs: 8/9
Firewire Port
Controller Boards
2 x (FPGA + QLA)
IDs: 4/5
dVRK
Motors
Sensors
Patient Side Manipulator 1 (PSM 1)
Motors
Sensors
Patient Side Manipulator 2 (PSM 2)
Motors
Sensors
Endoscopic Camera Manipulator (ECM)

Figure 3. Standard Hardware Communication

5.2 The Network Setup

In the network setup the Cloud Machine and the Local Machine are connected with an OpenVPN connection, hosted on the AWS instance. They communicate through a tunneled connection. The Ros Master is hosted on the local machine. The local client and the cloud service both connect to the Ros Master and each other respectively.

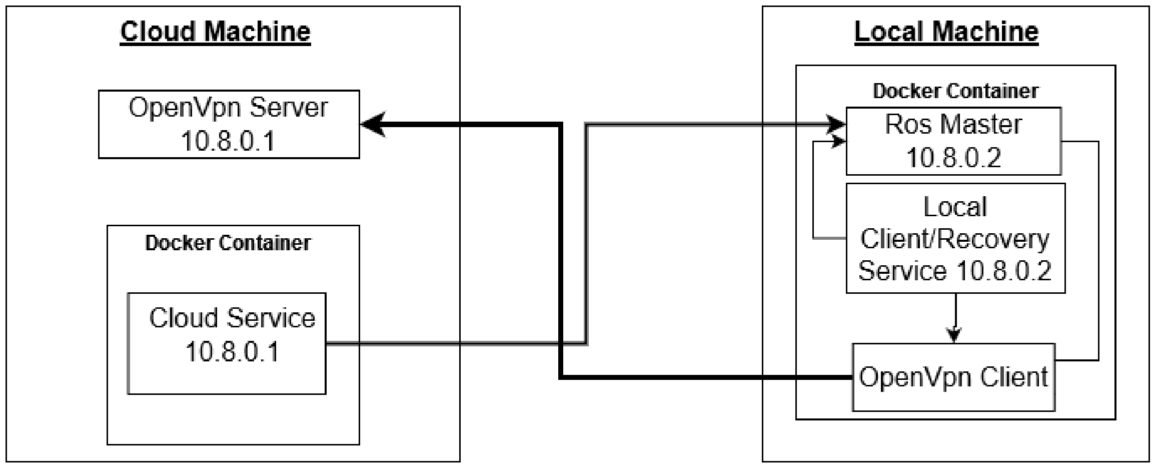


Figure 4. The Network Setup.

5.3 The Original Model

In the original model the ROS client-service architecture is used. The cloud functions as the client, and the local as the service. In this model, when the programs are both initialized the cloud client calls the local service when it has generated a sensing action. If the cloud client can not find a local service, it prints an error that the service was not found. If the local service is not called it waits until the program is killed or called.

The local container has only the coordinator and simulator. The coordinator interacts with the environment and maintains internal belief [14]. The coordinator in this project however, was a virtual environment designed to simulate a 2-dimensional Peg in Hole model. Therefore a simulator also runs on the local container. The cloud container contains an active sensing module, a task planner module, a coordinator and a virtual environment [14]. The cloud container generates the sensing action, and task action based on the feedback from the local. The virtual environment allows the cloud container to maintain belief and feedback about the environment so it can generate actions.

The following are the steps in the original model's primary operation [14].

1. The cloud container launches the ROS client and the local container launches the ROS service.
2. The cloud container calls the local container, with the request message set as the generated sensing action from its virtual environment.
3. The local container receives the sensing action, performs it, and gets the observation. It then sends the observation as a response message.
4. The cloud container receives the observation, updates its belief and then calculates the task action. It calls the local service with the task action as the request message.
5. The local service performs the task action and updates its belief. It then sends a continue signal to the cloud container.
6. The cloud updates its virtual environment and checks to see if a sensing action is allowed in the step. If not it performs nothing as a task action until a sensing action is allowed.

7. When a sensing action is allowed the cloud generates it and sends it as a request to the local service.
8. The virtual environments are updated in tandem, when they both have reached their goal or exceeded the maximum allowed amount of steps the simulation ends and the results are recorded.

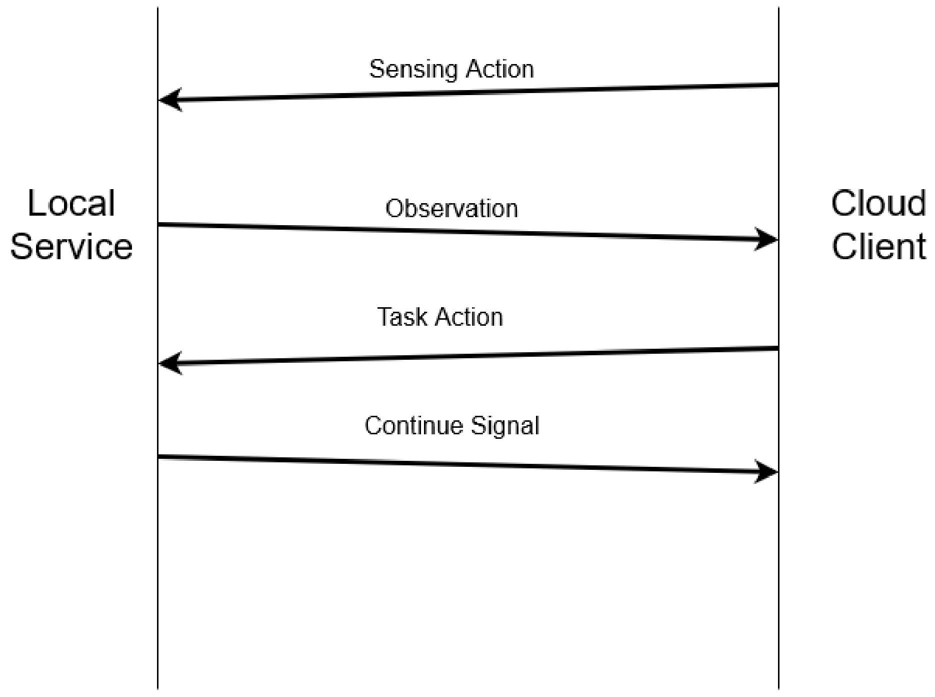


Figure 5. Original Model Communication.

The diagram above details the messages sent throughout this process. The cloud client sends the sensing action message as a request when it calls the local service.

5.4 The Revised Model

While the original setup was sufficient when a recovery mechanism was not needed, some changes needed to be made in order to support one. As is, ROS services have no method of detecting how long has passed since they've been called. ROS clients however, have a built-in method of detecting when a service is available, and waiting for a set amount of time if it is not, through the waitForExistence() function.

The ability to use this function was highly desirable for a system that was robust to recovery. To accommodate the use of this function, the local process was switched to a ROS client, and the cloud process was switched to a ROS service. The following are the updated steps in the revised model.

1. The cloud container launches the ROS client and the local container launches the ROS service.
2. The local container calls the cloud container, sending the observation as a request message.
3. The cloud container generates the sensing action and returns it to the local container.
4. The local container performs the sensing action, updates the belief, and sends the new observation as a request.
5. The cloud container receives the observation request. It updates the belief and calculates the task action.
6. The task action is sent from the cloud container as a response.
7. The local service performs the task action and updates its belief. It then sends a continue signal to the cloud container.

8. The cloud updates its virtual environment and checks to see if a sensing action is allowed in the step. If not it performs nothing as a task action until a sensing action is allowed.
9. When a sensing action is allowed the cloud generates it and sends it as a request to the local service.
10. The virtual environments are updated in tandem, when they both have reached their goal or exceeded the maximum allowed amount of steps the simulation ends and the results are recorded.

The communication process during standard operation in the new model is outlined below.

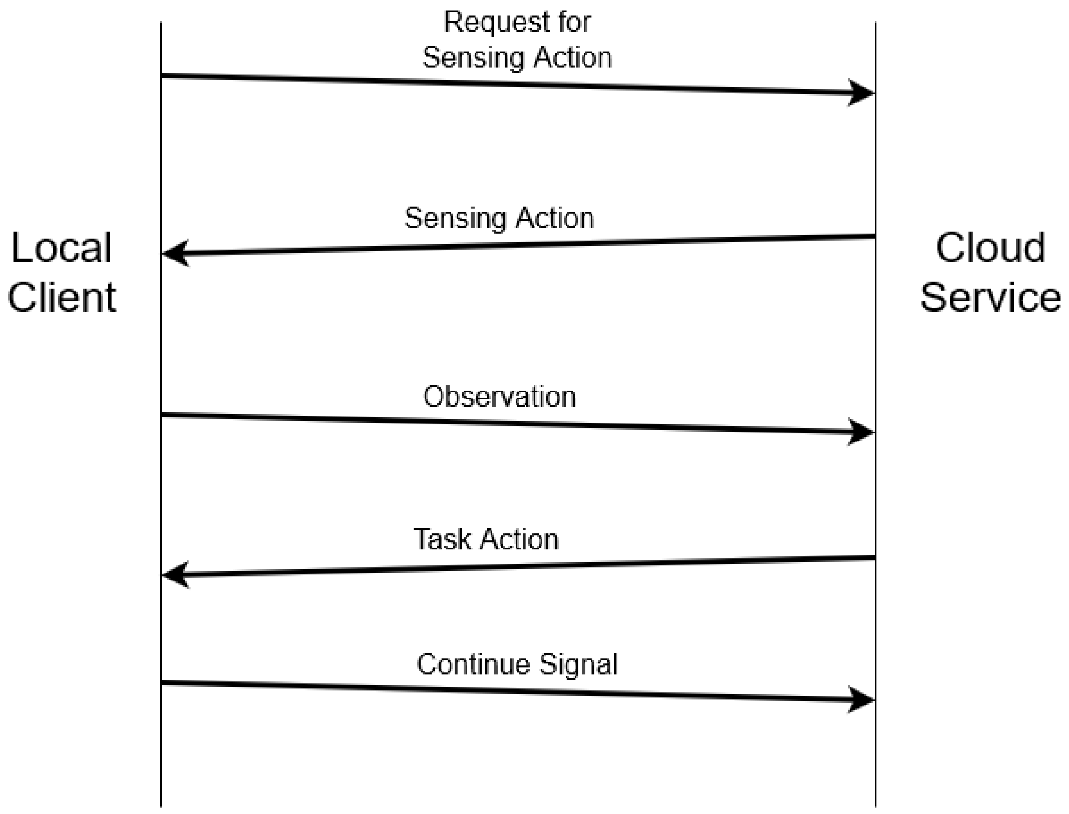


Figure 6. Revised Model Communication.

Additionally, the revised model has statuses to keep track of which state the program is in.

5.5 ROS WaitForExistence

Now that the service and client have been switched, the local container is able to take advantage of the in-built ROS function waitForExistence(). WaitForExistence() is a blocking call that takes a timeout value and a service as a parameter. If the service it is waiting for exists, it returns True and continues normal operation. Otherwise it will block the program until either the service exists, or the timeout is exceeded.

When the timeout is exceeded it returns False. This return can then be handled accordingly by the recovery method. Depending on the status that the local program is in, different recovery methods are executed.

5.6 Statuses and Definitions

Statuses are a set of enums implemented in the code to keep track of what events have occurred in the local service. The status is changed depending on the events and affects which recovery process should be followed. The status transition diagram is shown below.

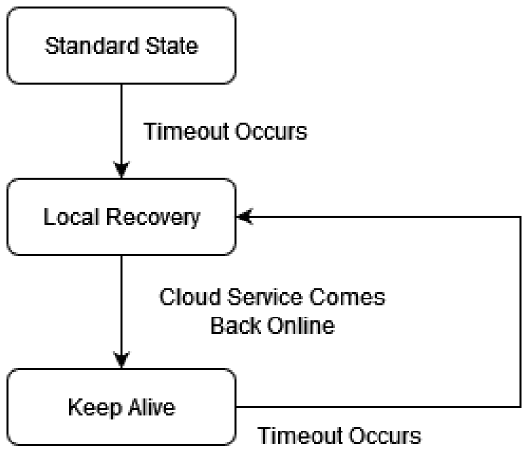


Figure 7. Status Transition Diagram.

1. Standard State:

   Standard operation indicates that the program is running as expected, and has not entered recovery mode at any point in the process. The cloud service has stayed active the whole time, and the client has been able to communicate with it. The local client starts in this state.

2. Local Recovery:

   Local recovery indicates that the cloud service is currently inaccessible. This status does not distinguish between whether it is the first time the local recovery has been activated, or the second. While in this state the local client attempts to call the cloud client each turn. If it cannot, it uses the local service. If it is able to then the status is switched to Keep Alive.

3. Keep Alive:

   Keep alive indicates that the client is currently able to access the cloud service, but that the local service has been launched previously. This status is used to indicate that the local service does not need to be relaunched, but simply called in case of another failure.

5.7 Standard Operation

Two clients are made when the local container is started. One client is connected to the service “cloud_active_sensing” which is the cloud service that traditionally runs the operation. A second client is connected to “cloud_active_sensing_recovery”, which is the service that is launched when the recovery mode is activated.

In standard mode all communication is run through the initial client. At the start of each sensing loop, the program uses the waitForExistence function to make sure that the service exists as expected.

This check is placed at the top of the while loop that occurs when a sensing action is permitted. After each message is sent, the program returns to the top of the while loop and waits for the service to exist. The timeout is set to five seconds to allot for a delay in starting the cloud service at the start of the program, and to allow time for small network errors. When the service is unavailable after the timeout, the recovery mechanism is launched.

The method for that is outlined below

Algorithm 1: Timeout Detection

```
1.  procedure: COMMUNICATION
2.  if step_number % sensing_interval == 0 && (status == Standard or keepAlive):
3.      while ros::ok
4.          if waitForExistence:
5.              request sensing action, send observation
6.              case sensing action received:
7.                  perform sensing action
8.                  request task action
9.              case task action received:
10.                 perform task action
11.                 request service continue
12.                 Break Loop
13.         else:
14.             Activate Recovery Service
```

The process to activate the recovery service is detailed in the section below.

5.8 Recovery Launch

The recovery program consists of another ROS program, run on the local container. It is not launched from the start to conserve resources, however in the case of a timeout a bash script is run that launches the program.

The program is the same as cloud service, except that it is run in the same container as the local client. In theory, on launch it would be sent the current state of the program and would resume operation from there. However, due to the constraints of the virtual environment being used, this particular recovery mechanism only functioned when the cloud service never launched. This problem is explored in more detail in simulation and results and future work.

When the recovery mechanism is launched the status is changed from standard operation to local recovery. While in local recovery the second client connected to "cloud_active_sensing_recovery" is used to communicate with the local service and get the sensing and task actions. The full operation is detailed in the following section.

5.9 Operation in Recovery

Two clients are made when the local container is started. One client is connected to the service "cloud_active_sensing" which is the cloud service that traditionally runs the operation. A second client is connected to "cloud_active_sensing_recovery", which is the service that is launched when the recovery mode is activated.

The communication structure for local client and local service mirrors the one for local client and cloud service, and is outlined in the diagram below.

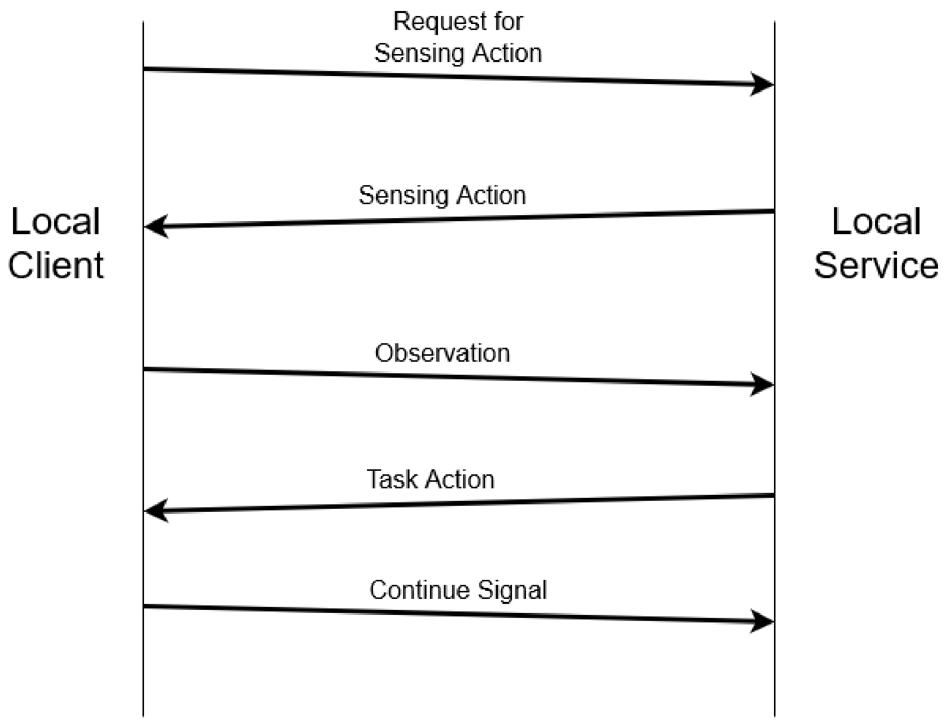


Figure 8. Communication in Recovery Mode.

5.10 Cloud Service Detection

Before each call is made with the second client connected to the recovery service, an attempt is made to call the cloud service. The call function in ROS allows for a client to try and call the service, if the service is not available the call function will return false. This is not used for the timeout detection method as it is instantaneous, and has no room for a delay in the service. However, when the client is already running in recovery mode, this is an effective way to see if the cloud service is available again.

If the cloud service is available, the status is switched to Keep Alive, and the cloud service is called again with a recovery flag activated. The recovery flag lets the cloud service know that the backup has been running, and the cloud service is passed the current state with the recovery flag.

Below is an outline of the method that has the cloud detection in local recovery.

Algorithm Two: Detect Cloud Service in Recovery

```
procedure: RECOVERYCOMMUNICATION
    15. if step_number % sensing_interval == 0 && (status ==
        localRecovery):
    16.     while ros::ok
    17.         if client.call(srv):
    18.             status = keepAlive
    19.         else:
    20.             client2 request sensing action, send observation
    21.             case sensing action received:
    22.                 perform sensing action
    23.                 client2 request task action
    24.             case task action received:
    25.                 perform task action
    26.                 client2 request service continue
    27.                 Break Loop
```

5.11 Operation in Keep Alive

Keep Alive operation follows the standard operation with the exception of the recovery mechanism. If the waitForExistence timeout is exceeded, the status is switched to localRecovery with the recovery flag activated and the active client is switched from client to client2. The operation then follows the recovery operation.

Chapter 6 Simulation and Results

6.1 Peg in Hole 2D Problem

This project used the peg in hole 2D problem to test the recovery mechanisms. The peg in hole problem is a traditional robotics problem, in which the robot tries to manipulate a peg into an appropriately sized hole, and drop the peg in the hole. The goal is considered reached if the peg successfully enters the hole, and not reached if it does not [14].

This problem is appropriate for a continuous space approximation, and is able to use the action entropy method developed by Greigarn [13]. The 2D space is used in this simulation to simplify the problem and focus testing on the recovery mechanism.

Let x represent the location of the peg, and let Pw and Ph represent the width and height respectively. The state of the peg at time t is represented by the x and y coordinates of the peg. The action is initially set to u = 0 when the simulation is started.

First, u will be set to approach the hole on the x-axis. The policy will calculate the peg's estimated location, if it is less than half of the width of the peg from the hole, the action will tell the peg to go to that location. When it reaches the location a new action is generated. If the X1 location is acceptable, then the X3, or angle is generated and compared. Finally, the X2 location is generated to drop the peg into the hole.

The belief is generated with a Gaussian distribution. The sensing action selects the direction of state space that should be observed to determine where the peg goes.

$$x_t = x_{t-1} + u_t min(1, \frac{a}{||u_t}) + n_u . \quad (5)$$

The noise is generated following a Gaussian distribution as well.

Since this project builds on Liu's work, we use the same settings for the Peg in Hole 2D simulation, with a few small changes. The simulation is repeated ten times in our experiment, and the maximum number of steps is set to 700 steps. If the simulation fails to reach the goal state within 700 attempts, that trial is considered to be a failure. If the simulation reaches the goal state, the trial is a success.

The following table represents the model configurations, which are stored and loaded from a YAML file [11].

state_size	3
peg_width	1
peg_height	2
hole_tolerance	0.1
init_mean	[4, 2, 0]
motion_cov	[2,1,1.57]
sensing_cov	.001

collision_tol	.001
num_trials	10
max_steps	700

Table 1. List of Configurations in YAML

6.2 Cloud Sensing ActionTime

Below are the average times for the calculation of the sensing action when both the cloud and local were operational for the entire run time of the program. The particles represent how many particles were used in each calculation, and the sensing action time is in seconds. The sensing action time is how long it takes from when the service receives the request for a sensing action to when it has finished calculating the desired sensing action.

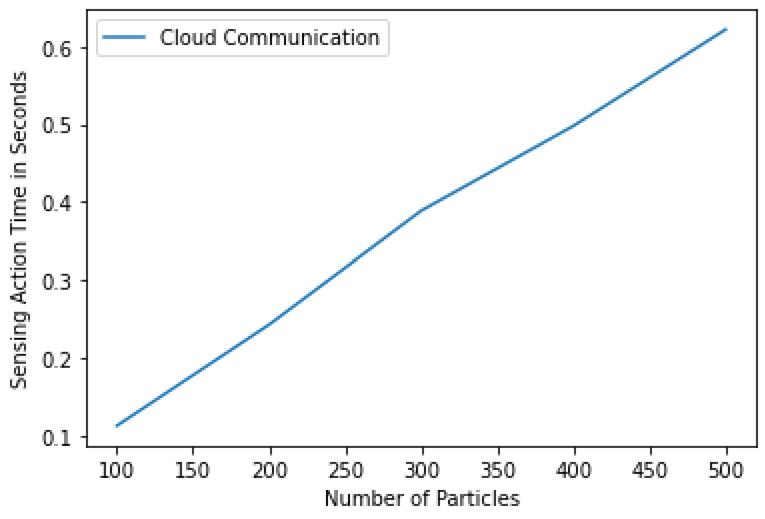


Figure 9. Cloud Active Sensing Time

6.3 Local Sensing Action Time

Below are the sensing action times for the local communication, when the cloud service never came up and the simulation was run entirely in recovery mode. The data is represented in seconds.

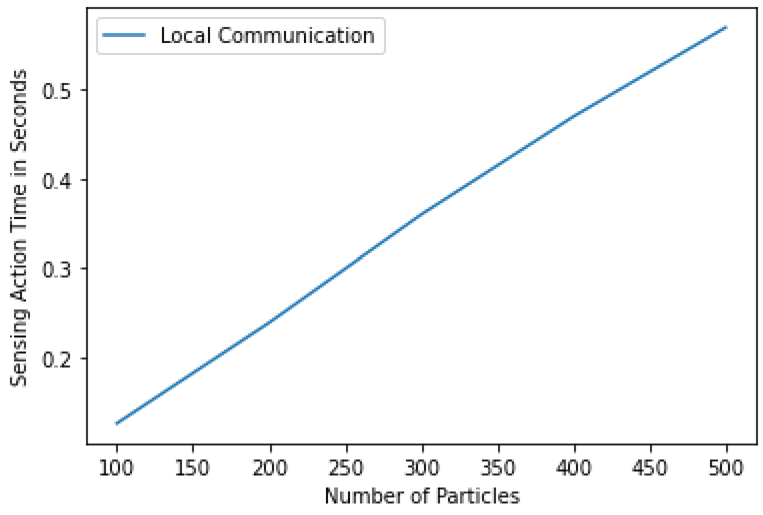


Figure 10. Local Sensing Action Time.

6.4 Side by Side Sensing Action Times

The following graph is a side by side comparison of the local and cloud computation sensing action times. The time is represented in seconds.

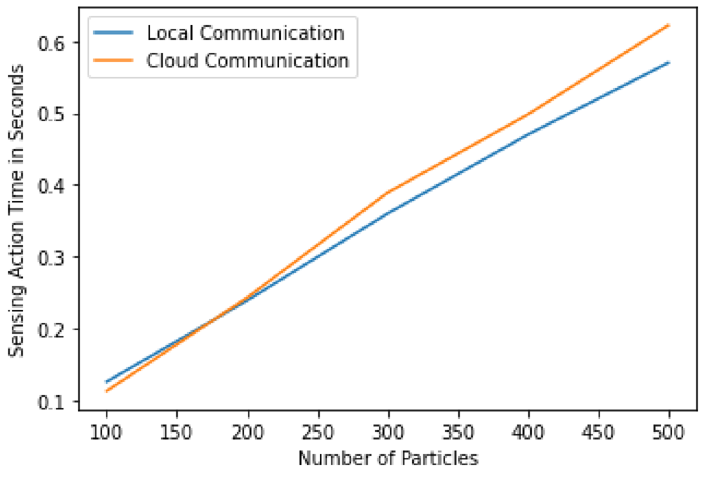


Figure 11. Compared Sensing Action Times.

6.5 Round Trip Time Comparison

The round trip time is the time it takes for a sensing action to be requested and received, minus the time to calculate the sensing action. This is to get a measurement of the time it takes for a message to travel from the local container to the cloud, or from one local container to the other program in the local container. The round trip times of both scenarios are shown below.

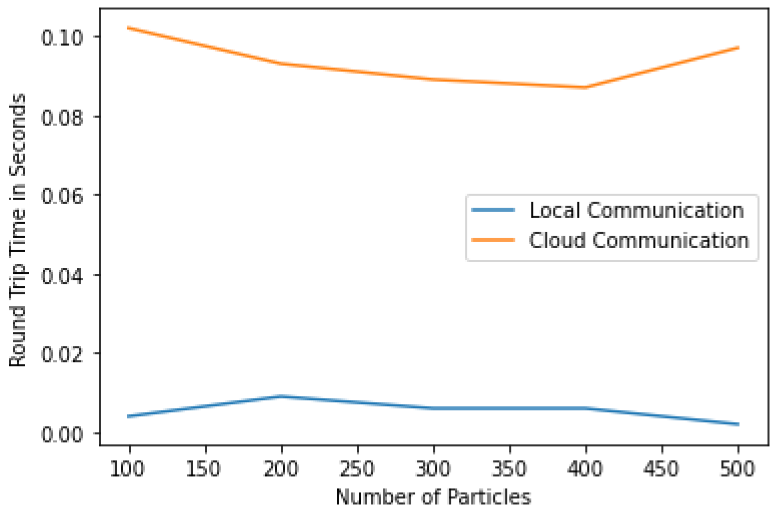


Figure 12. Round Trip Time in Seconds

.

6.6 Success Comparison

Below is the graph representing the number of successful trials out of ten for each of the simulations. Only one line can be seen on the graph because they had the same number of successful trials out of ten.

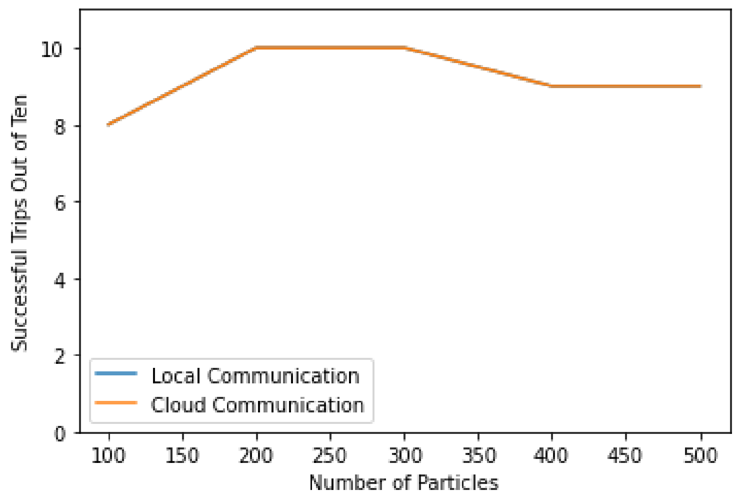


Figure 13. Successful Trials.

6.7 Time to Launch Local Recovery

Below is the time it takes for the local to come up when the cloud service is running and fails during the operation. In this measurement the sensing action is not calculated prior to the response time being sent, a response time is sent as soon as the service is called by the local. This was with a timeout of five seconds applied to the waitForExistence. The five second timeout is adjustable depending on the desired wait time, and so is shown on the graph as a baseline.

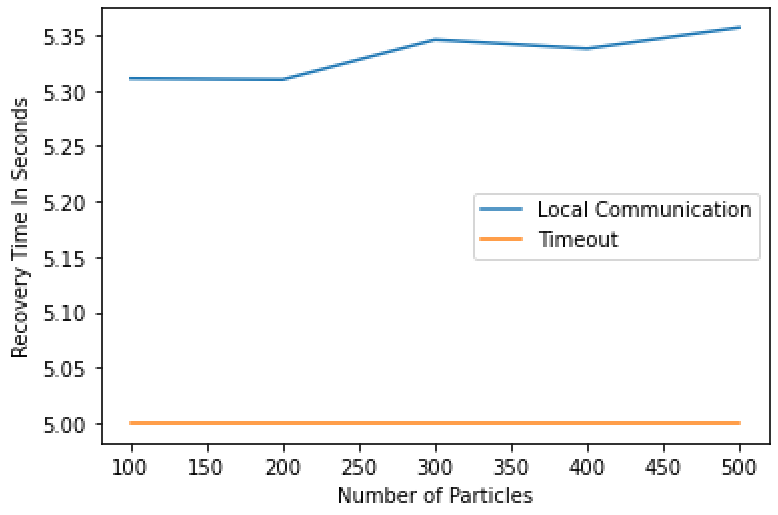


Figure 14. Time to Launch Local Recovery.

6.8 Time to Call Relaunched Cloud Service

The following is the time from when the cloud service was first rebooted to when it was first called by the local client. This timing indicates how long it takes the local client to realize that the new service exists, and switch operation over to it.

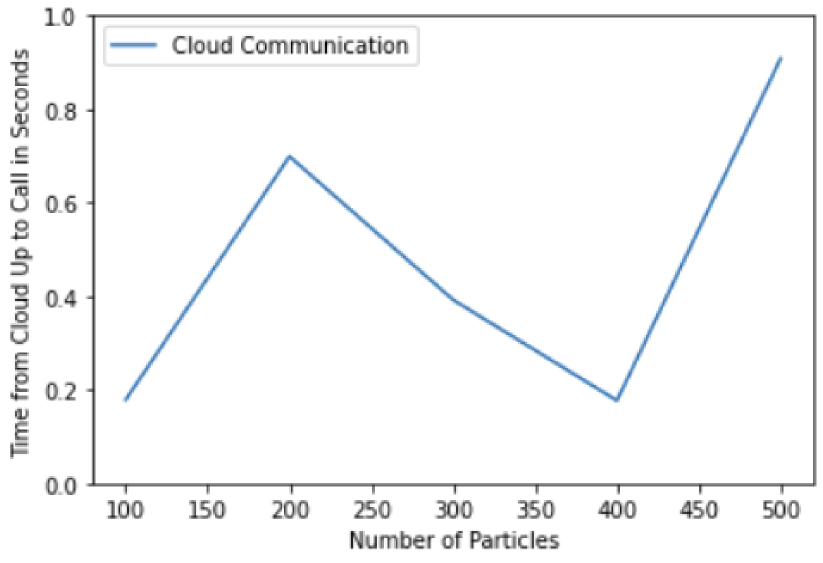


Figure 15. Time to Call Cloud Service.

Chapter 7 Discussion

7.1 Analysis of Sensing Action Time

When comparing the sensing action calculation times between the local and cloud, there is not a noticeable improvement when using the cloud service over the local. The only case in which the average sensing action calculation was faster on the cloud versus on the local was in the case of 100 particles, and it was only a speed up of 0.013 seconds.

If there is a sufficiently powerful local machine, the cloud does not provide a significant speedup in the time, and even took more time than the local machine. The local machine in this case had an i7-8700 CPU @ 3.20GHz × 12 and was running Ubuntu 22.04. It was able to be entirely dedicated to the task of running the local calculations, and the CPUs it uses are of consistent quality. Liu's work found that the AWS CPUs are often of non-consistent quality, which could lead to the timing mismatch [14].

These timing results are similar to the results of Liu's model [14]. This shows that the changed methodology does not cause an undue increase in the time taken to calculate the sensing action.

7.2 Analysis of the Round Trip Time

The round trip times varied dramatically between the all in recovery method and the cloud and local communication method. When run entirely on the same machine, the round trip time never went above six milliseconds, and when the local and cloud were communicating the round trip time never went below eighty seven milliseconds. This can largely be attributed to the communication gap between the AWS web service and the local computer. Additionally, while OpenVPN was able to run with a UDP connection, the ROS messages had to be sent with TCPROS, which increased the round trip time considerably.

7.3 Rewards Analysis

In Figure 17 it is clear that both the Cloud and Local share the same number of successful trials, as only one of the lines is visible. This is to be expected as they are run on the same situation with the same number of particles, and thus should come to the same results. In testing this matched up as expected, showing no difference between the overall success performance of the Local and Cloud Simulations.

Increasing the number of particles improved the overall performance from the initial number of eight successes. Once the particles exceeded 300 however, the performance ceased to increase and went down to nine successes. While fewer trials were performed since the focus of this project was on recovery, the percentage success rate is consistent with Liu's results. A decrease in performance is not seen with the change in methodology.

7.4 Analysis of Recovery Launch Time

The timeout is adjustable depending on the desires of the operator of the system. For the purposes of this experiment it was set at five seconds. The client would wait five seconds for the cloud service to become available before either activating the local service or switching to using the local service depending on the state.

Due to the adjustability of the five second timeout that timeout will not be included when considering how long the recovery takes to launch. The backup never took more than 370 milliseconds from when it was booted to when it became available. This is an acceptable amount of time for a recovery mechanism, in particular when the alternative is waiting a potentially indefinite amount of time for the cloud service to return.

There was little variance in how long it took the recovery method to be available, and so this method of recovery proved to be scalable at least up to 500 particles.

7.5 Analysis of Time to Detect Cloud

The time from when the cloud service became available and was first called is shown in the function. This time varied more dramatically between the number of particles, but this variance can be explained by the service returning at different points in the local's operation.
When the program does not yet need a sensing action it waits to call the service, so if the cloud is started while it is performing a task action it will take longer to request the cloud service.

Regardless of the variance, all of the recovery times are acceptable. It never takes more than a second for the local client to find the cloud service once it's initiated, and from there it can switch back and forth as needed with ease. Once the cloud service is available it is the only service utilized.

Chapter 8 Conclusion and Future Work

8.1 Conclusion

Despite the minimal or non-existent speedup when using the cloud versus running the processes separately on a local machine, having a cloud alternative is valuable. Cloud systems are more scalable than hardware, and thus can be upgraded more cheaply if the local machine is found to be insufficient. Having a backup for the cloud is still a worthwhile investment.

This project restructured the existing communication of Liu's OpenMP action-sensing calculation in order for it to support a timeout detection for when the cloud service failed.We did this by switching which process used the ROS service and client in order to utilize the ROS client waitForExistence function.

States were implemented to track which service the local should communicate with, and which recovery mechanism should be launched. The time to recovery with each method was measured, and found to be acceptable.

Future work has three main directions, changing the virtual simulation so the process can be picked up from the current state, implementing UDP ROS communication with services, and creating a more resilient ROS master.

8.2 Changes to the Simulation Environment

Currently the simulator uses a virtual environment on both the local container and the cloud container, with the local container’s environment standing in for the robot. However, the way that the simulations are coded on both sides means that they have an additional amount of noise coded in. When they’re started at the same time with the same seed, that random noise matches up on both ends and the simulators are able to stay in sync.

If one simulator is started later than the other however, and generates a different number of sensing actions, task actions, or observations, the random noise patterns do not match up, and the simulators fall out of sync. This means that the recovery mechanism is unable to be fully implemented, because even if the current state is passed to the virtual environment on the cloud and implemented, the noise will keep the states from being correct. This is initially a small displacement, but as the cloud generates sensing and task actions from this noise, the states diverge further. Fixing this is essential to a functional recovery mechanism.

8.3 Changing from TCP ROS to UDP ROS

The current project uses TCP connections to pass messages between the Service and Client in ROS. While TCP is more reliable than UDP, it is much slower than a UDP connection. Since this project is designed with the hope of eventually being able to plan in real time, those delays are untenable.

A UDPROS protocol is currently in development, however at the time of this project it was only functional for publishers and subscribers, and not for clients and services [15]. Documentation exists for a UDPROS protocol for clients and services, but the page is simply an outline of a plan to implement it [15]. Test cases are described, but do not seem to have been executed, and there are no instructions on how to utilize it [16]. A fully implemented and tested version of ROSUDP for services and clients would improve the speed of network communication in the future.

8.4 Improving the Resilience of ROS Master

The ROS master is a key part of the communication system used in this project. It must be running in order for the other nodes to start and function. It provides naming and registration services, as well as tracking publishers and subscribers to services. All other nodes rely on the ROS master.. No programs will run without it. The ROS infrastructure uses a Bridge design pattern, allowing it to use multiple different types of implementation on an abstract level. This is somewhat handled when the processes are made into packages and run, but ROS master is still essential for packages built with different underlying code to communicate.

Due to the essential nature of ROS master for communication, having a way to detect the failure and restart it is crucial. Unfortunately, the current structure of the ROS master makes this difficult. If the ROS master crashes, a new one must be started. When the new master is started, all of the current programs have to be restarted as well so they can be registered with the new master.

This process is functionally a system restart, even with the state saved. The time cost to fully restart the system, as well as the inability to predict the behavior of the robot on the system shutdown, lead to this being an unacceptable solution. Future work that focused on adding resilience to the ROS master architecture would be a huge improvement in the overall reliability of the system.

Reference

[1]. Y. Liu and Y. Xu, "Summary of Cloud Robot Research," 2019 25th International Conference on Automation and Computing (ICAC), Lancaster, UK, 2019, pp. 1-5, doi: 10.23919/IConAC.2019.8895254.

[2]. Tian Guohui, Xu Yaxiong, “Cloud Robotics: concept, architectures and key technologies” Journal of ShanDong University, (Engineering Science), vol 44, no. 6, pp. 46-54, Dec. 2014.

[3]. Kehoe, Ben, et al. "A survey of research on cloud robotics and automation." IEEE Transactions on automation science and engineering 12.2, 2015: 398-409.

[4]. Juve, G., Deelman, E., Berriman, G.B., et all, “An Evaluation of the Cost and Performance of Scientific Workflows on Amazon EC2.”J Grid Computing 10, 5-21, 2013.

[5].“What is Ros?,” Ubuntu, https://ubuntu.com/robotics/what-is-ros (accessed May 21, 2023).

[6]. Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." ICRA workshop on open source software. Vol. 3. No. 3.2. 2009.

[7]. “TCPROS”, ROS.org, http://wiki.ros.org/ROS/TCPROS, April 15, 2013. (accessed May 22, 2023).

[8]. “Master”, ROS.org, http://wiki.ros.org/Master, January 15, 2018. (accessed May 22, 2023).

[9]. “What is a Container”, docker, https://www.docker.com/resources/what-container/. (accessed May 22, 2023).

[10]. “What is a VPN”, OpenVPN, https://openvpn.net/what-is-a-vpn/. (accessed May 22, 2023).

[11]. “Overview of Amazon Web Services”, Amazon Web Services, https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html, April 15, 2023. (accessed May 22, 2023).

[12]. “What is Amazon EC2”, Amazon Web Services, https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html (accessed May 22, 2023).

[13]. Greigarn, Tipakorn, Michael S. Branicky, and M. Cenk Çavuşoğlu,, "Task-Oriented Active Sensing via Action Entropy Minimization.", IEEE Access 7 (2019): 135413-135426.

[14]. Liu, Yuwei. "OpenMP based Action Entropy Active Sensing in Cloud Computing." Master's thesis, Case Western Reserve University, 2020. http://rave.ohiolink.edu/etdc/view?acc_num=case1584809369789769

[15]. “Publishers and Subscribers”, ROS.org, http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers#Transport_Hints, April 10, 2018. (accessed May 22, 2023).

[16]. “UDPROS”, ROS.org, http://wiki.ros.org/ROS/UDPROS, April 20, 2013. (accessed May 22, 2023).