BUILDING INTELLIGENT AUTONOMOUS AGENTS AND MULTI AGENTS USING THE FETCH.AI DECENTRALISED OPEN ECONOMIC FRAMEWORK.

Review

MawaMaverick

Abstract

It is eminent that artificial intelligence and distributed technologies will arbitrarily play a pivotal role in the emerging 4rth industrial revolution. Data has solidified its place as the fuel of the future economy, which means that efficient and automated data handling techniques have to be developed and/or improved upon, in order to facilitate all new shapes and forms in which data will be generated, collected, consumed, transacted and utilized. In this paper we review a platform that is being built to harness data and provide a viable use-case by leveraging an **agent-**based system of communication by combining three fundamental concepts at its core i.e. **Decentralisation**, **Artificial Intelligence (AI)** and **agents** . Before we get into it, let us look at the fundamental characteristics of these technologies.

- **AI** – This is fundamentally probabilistic and employs ever changing algorithms that make learned guesses at reality. Any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals
- **Block chain and Distributed ledger technology** – fundamentally employs deterministic, permanent algorithms and cryptography to record reality.
- **Agent** – An autonomous and intelligent entity that acts, while directing its activities towards a specific goal within an environment through sensors and consequent actuators.

As we move into a more data driven economy, we find that data is generally becoming more valuable as its use-cases increase and diversify.

In this document we are going to review a proposed method of data computing approach, Agent Oriented Programming. This computing paradigm promotes and necessitates a collective view of computation in which a Multi Agent System allows for multiple **agents** to interact with one another in a continuous and autonomous manner without the need for human guidance or intervention.

Table of Contents

Acronyms

PoW – Proof of Work
PoS – Proof of Stake

DPoS – Delegated Proof of Stake

ZK-SNARKs – Zero Knowledge Succinct Non-Interactive Argument of Knowledge

PoC – Proof of Capacity

PoAuthority – Proof of Authority

PoB – Proof of Burn

BFT – Byzantine fault Tolerance

DBFT – Delegated Byzantine Fault Tolerance

DAG – Directed Acyclic Graphs, etc.

Svm – Support Vector Machine

uPOW – Useful Proof of Work. (Ensures trust and integrity of the network.

DAG – Direct Acyclic graph

OEF – open Economic Framework

soef – simple OEF

MAS – Multi Agent Systems

AEA – Autonomous Economic Agent

INTRODUCTION

There are a lot of privacy and geopolitical concerns around data management and who controls what. With an ever increasing concern about the centralisation of data amongst “Big-Tech” companies, there is a growing movement in the sphere of decentralised data consumption, which involves harnessing node based systems, in the sense that data control and storage should ideally move away from centrally controlled hubs and servers, and onto distributed and decentralized node systems.

Cryptography such as private key encryption and public key access has accelerated the move towards this approach of data usage, by bringing security and scalability. As a result, data ownership is slowly moving back to the individual. This however brings with it technical hindrances that many companies, individuals and countries just can't afford to deal with due to a lack of technical know-how, finances and time to correctly handle and manage. Therefore there is a need for technical solutions that would help skip over such technical huddles while data ownership is guaranteed and privacy and security are maintained. This can only be guaranteed by harnessing technologies that re-allocate these fundamental roles from erroneous human third parties, to incorruptible and trust-less consensus algorithms that are run by decentralised and publicly distributed computing systems.

All this is made smarter by creating frameworks that enable inter-operability between decentralised technologies and Artificial intelligence. By combining the two, it implies that sensitive data can be moved and shared between different parties without compromising privacy and security, whilst access to a particular set of data would only be available to an entity that possess the **key** required to decrypt it. Various encryption algorithms such as Secure Hash algorithm like SHA 256, Elliptic Curve Digital Signature algorithm (ECDSA) are used to guarantee encryption within permission-less decentralised and distributed public ledgers such as bitcoin and Ethereum, respectively.

For users of a public permission-less system to agree on the validity of the encrypted data being moved around, consensus-based algorithms are used. Consensus algorithms such as Proof of Work, and Proof of Stake etc. A consensus algorithm may be referred to as method used to reach an agreement on a single data value among distributed processes or systems or the current state of a distributed system. Primarily they help to achieve reliability in a network that involves multiple nodes that contain the same or similar information.

Computer science is evolving in a very complex and sophisticated way that is eventually meant to give way to highly intelligent self-governing, self-sustaining systems that work to simplify the day to day life of humans and machines in the physical and virtual world. To facilitate these machine to machine, and human to machine interactions, self-conscientious artificial beings have to be created, and the major direction is pointing towards **Autonomous agents** that collectively work continuously and autonomously in an environment in which other processes take place and other agents exist. Agents ideally have to be goal driven, with the ultimate aim of making the best informed decisions on behalf of their owners (be it human or other machines), based on cleverly designed hierarchical modular structures that employ philosophical **principles such as ontology, belief, desire, intent, abstraction, objectivity, semantics and social ability.**

What is an Autonomous agent?

To answer this, we must briefly examine the history of agents and when the idea was first proposed. Agents go back as far as 1948 with an instrument called the Turing machine that was designed by Alan Turing. The computing machine facilitated an environment containing two agents. One agent generated **enigmas** and the other **solved them.** Situated and flexible, the agents would receive input from the environments and then independently act and offer feedback. This was a profound definition of autonomy. It is imperative to note that

Turing also hypothesised that **cryptography** could be **intelligent machines' most rewarding task.**

Early Autonomous agents were also proposed in **Claude Shannon's "A Mathematical Theory of Communication"** *[Shannon, 1948]*. It is evident that crypto intelligence provided the basic template for Shannon's watershed contributions to early artificial intelligence. [1]

When two or more agents work collectively in a dynamic environment to make decisions, they are referred to as a Multi Agent System (MAS). The MAS paradigm appears in 1980 with its core specificity revolving around collective behaviour. Initially the work in this field (1970-1980) focused on distributed problem solving where knowledge and processing are distributed, but the **control is centralised,** also to be noted is that these systems are also ad hoc in the sense that they cannot be repurposed for differing use-cases. A second phase of agents is seen around 1980-1990 where the idea of re-usability is much more prevalent, and in the third phase from 1990-2000, we see a focus that is mainly on agent to agent interactions. Currently. the focus is on goal based, self-organising systems. Where just like standardisations like the ISO, ISO-20022, there is a standard known as the **Foundation of Intelligent Physical Agents** or FIPA for short. FIPA specifications represent a collection of standards which are intended to promote the interoperation of heterogeneous agents and the services that they can represent.

MASs are not only going to change the efficiency of technology in general, but also represent a novel general-purpose phenomenon in software development. The advent of agent-based computing promotes design and development of autonomous software entities with well organised hierarchical protocols and languages. [2]

Examples of methods that have been proposed over the years as a way of managing agent and agent-based systems:

In 1990, an Agent Oriented Language known as **Agent0** that was introduced by Yoav Shoham was the first programming language that was explicitly developed for programming agents. His idea was to introduce a language whose computation was based on a societal view of computation. The main aspect of this language is that it takes seriously the idea of 'intentional' notions that directly attribute their assumptions on the mental states of agents; States such beliefs, desires, commitment and intentions, with the end goal of the agents leveraging these characteristics to accomplishing their tasks.

In Agent0 an agent ideally has 4 components i.e.

- a set of capabilities (things the agent can do)
- a set of initial beliefs.
- as set of initial commitments (things the agent will do), and
- a set of commitment rules(This is the programming part). How agents are told how to generate new commitments.

Commitment rules Determine how the agent acts: i.e. Each commitment rule contains

  - message condition

- a mental condition, and
- an action

Other models or methods proposed include but not limited to:

MASON:

Developed at George Mason University's Evolutionary Computation Laboratory. MASON is short for Multi Agent Simulation of Neighbourhoods (or Networks), and it is a multi-agent simulation environment developed in the **Java** programming language.

JADE:

Java Agent Development Framework or JADE is a software framework used for the development of intelligent agents. It is implemented in Java. JADE supports the interoperability with FIPA (Foundation for intelligent Physical Agents) and **ACL (Agent Communication Language).**

Just like ACL also known as FIPA-ACL is a standard proposed by FIPA, the other is **KQML** (Knowledge Query and Manipulation Language). Both rely on a speech act theory developed by **John Rogers Searle in the 1960s.**

REPAST:

The Recursive Porous Agent Simulation Toolkit is a widely used free and open-source, cross-platform agent-based modelling and simulation toolkit. Repast also has inbuilt adaptive features such as genetic algorithms and regression.

NetLogo:

This is a programming language and integrated development environment (IDE) for agent-based modelling. NetLogo is implemented in Scala and Java , however it is mainly for the purpose of creating agent simulations.

MESA:

This is a Python-based alternative to NetLogo, Repast and/or MASON for agent-based modelling. It requires setting up a virtual environment and creating an agent using class and object style programming.

In this Paper we focus on exploring a completely new approach to agent-based development, an approach being developed by a company called **Fetch.AI**. The approach is new in the sense that it implements the philosophical proposition that is historically laid out by the most prominent figures that have developed a significant amount of research around agent-based systems.

Fetch.AI. It is a Cambridge-based artificial intelligence lab that is building a decentralised machine learning platform that is based on a distributed ledger. Fetch.AI is creating an ecosystem that enables and facilitates an agent based framework that encompasses the following characteristics, but not limited to them: **Collective intelligent agents, Security, decentralisation, open-source, free market, transaction/negotiation, scalability, decision making, Internet of Things (IOT), transport and decentralised deliver network**. All this is possible through their novel innovative solutions that enable intelligent

agents known as **Autonomous Economic agents** to be able to perform useful tasks on behalf of their owners whilst residing in an Open Economic framework that's built on top of the **Fetch smart ledger**.

Multi Agent Systems go hand in hand with machine learning and artificial intelligence, especially in field of **Reinforcement Learning** – where the general idea is to have a system that recognises the concept of agents, environment, and action. An agent perceives its surroundings: an environment is the one within which an agent interacts and acts upon. The main goal in reinforcement learning is to find the best possible route to a specified problem.

Fetch.AI

Fetch.AI is a blockchain based solution that seeks to merge Machine Learning and blockchain-technology. The essence of Fetch.Ai is to create value within data. Through an Open framework, a free marketplace exists where **Autonomous Economic Agents (AEAs)** work together and collectively, to not only make sense to themselves, but as result they create value for their owners and users.

Blockchain and distributed technology as hinted earlier, leverage mathematically based permission-less consensus and cryptographic algorithms with the (SHA-1, SHA-2 and SHA-256) being the most popular cryptographic algorithm.

More examples of consensus algorithms are listed below.

- PoW – Proof of Work
- PoS – Proof of Stake
- DPoS – Delegated Proof of Stake
- ZK-SNARKs – Zero Knowledge Succinct Non-Interactive Argument of Knowledge
- PoC – Proof of Capacity
- PoAuthority – Proof of Authority
- PoB – Proof of Burn
- BFT – Byzantine fault Tolerance
- DBFT – Delegated Byzantine Fault Tolerance
- DAG – Directed Acyclic Graphs, etc.

Decentralised open and permission-less distributed and blockchain technologies are known to have a major problem. i.e. the failure to exist in a state where the following three scenarios are simultaneously satisfied.

- Decentralized,
- Scalability and,
- Security

This problem is known as the **blockchain trilemma. Note:** Blockchain is a term that has been adopted to mean decentralised technologies, but not all public decentralised ledgers use blockchain technology. The blockchain trilemma refers to the generally accepted idea that it is not possible to scale a public blockchain network without compromising either security, decentralization, or both

There is no known blockchain algorithm that has fully satisfied this issue, for there is always a compromise of something in favour of the other.

Security – This essentially is the level of defensibility a blockchain has against attacks from external sources. A secure protocol needs to be resilient in the short term and immutable in the long term. Examples of attacks that can occur because of compromised security.

- 51% attack. This can occur if one entity controls a very large proportion of the resources of a given ledger. These resources could be tokens, hash rate/mining power.
- Sybil attack. If an entity forges multiple identities on a system to control a significant stake in ownership and decision making
- Penny spend attack – an entity that attacks a protocol with low value transactions to stop the network from running.
- DDoS – Distributed Denial of Service attack. The network could be overloaded with network with malicious transactions.
- Collusion – Where one or more entities or nodes decide to perform a malicious operation on the network

Scalability – This is the protocol's ability to handle a growing and large volume activity happening on it. Scalability is highly important because it dictates the network's capacity. Especially if the network intends to have millions of activities happening constantly and simultaneously

Decentralization – This reflects the level to which transactions between agents are possible and effective without the control or authorization of third-party groups or individuals. The more decentralised the protocol, the harder it is for a single party to take down.

Decentralization also comes with some other perks such as, higher number of participants, higher system fault-tolerance and security.

People are constantly coming up with ways to scale public distributed technologies. Methods such as.

- Sharding – having transactions in a parallel format
- lightning network – this solution does not scale the actual ledger itself. It is what is referred to as off-chain scaling.
- DAG – Using Direct Acyclic graph algorithm

For the issue of scalability and security combined, the **Fetch.AI** blockchain employs the following.

- **uPOW** – Useful Proof of Work. (Ensures trust and integrity of the network.
- **DAG** – Direct Acyclic graph. This has been used as an inspiration for parallel execution and transactions reference while requiring the strict ordering found in a transaction chain.
- **Sharding** – A resource lane is introduced through Sharding. Unlike traditional Sharding, in the fetch.ai system, a transaction may be assigned to several different resource lanes simultaneously.

Some of the features that do enable this Fetch.AI to be able to facilitate an environment that accommodates a very large number of agents are.

- Blockchain Sharding to increase concurrency and scalability
- The ability to program smart contracts with the native language called ETCH. This makes if possible, to develop programs that are compatible with Machine learning and AI capabilities as well as be able to communicate these capabilities with other agents.
- An Open Economic Framework (OEF) was decided native to the Fetch smart ledger. The OEF is a dynamic environment within which the agents reside and get input data.
- Support for fixed-point arithmetic to guarantee precision and determinism across all operations and transactions.

Mixing blockchain technology and agent-based systems opens up many possibilities and can indeed bring into reality truly decentralised autonomous worlds. Truly large-scale decentralized solutions can be built for the supply chain with a high performance blockchain that has large throughput.

Some of the advantages that the fetch ledger presents for engineers and developers to maximise its capabilities.

- 30K TPS (transactions per second)
- Fetch.ai has come up with a breakthrough solution to the ‘blockchain trilemma’ (i.e. security, scalability, decentralisation). Achieved by combining the consensus uPOW consensus algorithm and DAG technology with a novel breakthrough algorithm developed by fetch.ai called the Decentralised Random beacon.
- Interoperability with other ledgers such as Ethereum and Cosmos.

Fetch.ai can be summarised to have the following functionality.

Collective learning:

The combination of AI and blockchain helps in eliminating a lot of obstacles that have been in the way of data sharing. The use of **MAS (Multi Agent Systems)** also enables collective learning: This is the act by which **autonomous agents**, with competitive or complementary interests, increase their understanding of the state and behaviour of the **decentralized**

ecosystem they are connected to. Ideally, this will allow them to improve their solution to a problem.

Etch:

The native language in which smart contracts can be written with support for ML and AI.

FET:

This is the **cryptocurrency** that powers the Fetch.ai ecosystem.

Synergetic smart contracts:

Smart Contracts (SC) are computer programmes that allow developers to harness the potential of an underlying blockchain infrastructure through the automation and execution of a program or transaction protocol according to the legal terms and agreements of the contract. Synergetic Smart Contracts are extensions to the original concept of smart contracts enabling off-chain computation to be included in agreements involving multiple parties. Synergetic contracts enable a developer to run off-chain activities such as machine learning models onto the fetch smart ledger.

Elaboration of what an AEA is:

We mentioned **AEAs** earlier so let us try to elaborate on what they are and what they do. *"An AEA is referred to as an intelligent agent acting on an owner's behalf, with limited or no interference, and whose goal is to generate economic value for its owner*." In short, *"software that works for you".*

An **agent:** "*Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors."* [3].In this case agents act on behalf of someone or something else.

"An **agent** is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives" [4]

Economics: *" This is a branch of knowledge concerned with the production, consumption, and transfer of wealth."* Economics focuses on the behaviour and interactions of economic agents and how economies work.

Autonomy: "Ability to be Independent or self-govern". Essentially, "acting without constant interference from an owner"

An **AEA** is an entity with a well-defined goal such as:

- **buy x** for me when it is **cheapest**
- **sell y** for me to the **highest bidder**
- **if x** happens make sure I get **y done**

However, we must note, **not to** classify AEAs under the following categories because they are fundamentally different.

They are not.

- Artificial General intelligence (AGI). An AEA makes independent decision to reach a defined goal that directly involves an economic gain, unlike AGI
- APIs (Application Programming Interface), it is an interface that defines interactions between multiple software intermediaries. An API's purpose is already pre-defined.
- Smart contracts.

APIs (Application Programming Interface), it is an interface that defines interactions between multiple software intermediaries. An API's purpose is already pre-defined.

AEA Framework

The focus of an **AEA,** is to generate economic value.

AEAs learn do this by leveraging specialized software modules referred to as **skills.** AEAs autonomously acquire new skills, either through direct acquisition software modules or shared learning. e.g. If an AEA has a goal to acquire concert tickets at the cheapest price and already has the negotiation **skill** needed to achieve this. The agent will go ahead to search for sellers, and on encountering another agent selling said tickets via an auction, it would then autonomously acquire the necessary negotiating skill from another experienced agent without any necessary extra input from its owner.

The purpose of this paper is to further elaborate on what it means to create an Autonomous Economic Agent using the Fetch.AI **Open Economic Framework** (OEF).

Everything will be developed in a python-based development suite known as an **AEA Framework,** within which there are development infrastructure and tools that are deliberately designed to design compatible and interoperable agents. This framework is meant to work as well as how a modern-day web development framework would work.

AEAs achieve their goals with the help of the OEF - a search and discovery platform for agents that is developed by Fetch.AI - and using the Fetch.AI blockchain as a financial settlement layer. Third-party blockchains, such as Ethereum and cosmos may also allow AEA integration.

For the AEA to work, it must have building blocks or core elements. Foundational elements that define the agent's core philosophical and conscientious claims to its own behaviour and decision-making process.

- Envelope
- Protocol
- Connection
- Multiplexer
- Skill
- Main Loop

Envelope:

The envelope is design to be the medium that encompasses all the internal aspects of the agent's communication framework and capabilities. An envelope can be looked at as the vehicle that encompasses and holds the messages with 5 attributes

- **to** defines the destination address.
- **Sender** : defines the sender address
- **protocol_id** defines the id of the protocol. There are various protocols that fetch incorporates, like the Fetch ledger itself, the Ethereum protocol, cosmos, Nexus etc.
- **Message** : is a byte field which holds the message in **serialized form**. Serialization entails a process of translating a data structure or object into a format that can be stored in a file or memory buffer, or transmitted e.g. across a network connection link and reconstructed later to be used to create a semantically identical clone of the original object.
- **Optional[context]** :an optional field to specify routing information in a URI

Even if an agent is not an API, it has some fundamental similarities in the sense that it carries some predefined information that would be useful to the end user in whichever way they decide to implement it. We must remember that an agent executes a request if it wants to. It is not obliged by essence of its nature to accept and carry out whichever request comes its way, unlike an API.

Protocol:

Protocols define how messages are encoded for their transportation; As such, stipulating the basic rules with which an agent will adhere to, as per the message sequence. For instance, BUY and SELL or START and FINISH. The rules may be such that any message with SELL must be preceded with a buy and similar rules for FINISH.

The message class in the **protocols/base.py** module provides an abstract class with all the functionality that a derived **protocol** message class requires for a custom protocol, such as basic message generating and management functions and serialization details.

The basis for the implementation of a framework is sparked off by a default protocol called **default**. This thus provides the implementation fundamentals of an AEA protocol which includes a **DefaultMessage** class and a **DefaultSerialization** class that contains functions for managing serialization.

Additional protocols can be added as packages or generated with the **protocol generator**.

Messages can be protocol specific and when wrapped in envelops are sent and received amongst other agents and services via **connections**.

Connection:

The module **connections/base.py** contains the abstract class which defines a **connection.** A connection as a bridge to the SDK (Software Development Kit) or API to be wrapped, and is, where necessary, responsible for translating between the framework specific **Envelop** and its contained **message** and the external service or third-party protocol (e.g. HTTP). Just

like in APIs, where we have JSON interacting with HTTP heads on the server side or client side whilst sending **get, post, put, delete, and patch** requests. In JSON also serializable value data is used.

The framework provides one default connection called **stub**. This implements an I/O (Input/Output) reader and writer to send messages to the agent from a local file. Additional connections can be added as packages.

An AEA can run these connections via the **Multiplexer.**

Multiplexer:

The **multiplexer** is responsible for maintaining potentially multiple connections. It maintains an **InBox** and **OutBox**. These are queues for incoming and outgoing envelopes, respectively. They are also used to separate the main agent loop from the loop which runs the multiplexer.

Skills:

The skills are the c**ore focus** of the framework's extensibility. They are self-contained capabilities that AEAs can dynamically take on board, to expand their effectiveness in different situations.

A skill encapsulates implementations of the three abstract base classes **Handler**, **Behaviour**, **Model**, and is closely related with the abstract base class.

Expanding on what they do.

Handler:

Each skill has none, one or more Handler objects, each responsible for the registered messaging protocol. Handlers implement AEAs' **reactive behaviour**. **If the AEA understands the protocol referenced in a received Envelope, the Handler reacts appropriately to the corresponding message**. Each Handler is responsible for only one protocol. A Handler is also capable of dealing with internal messages.

NB: We also normally find the Call for Proposal (CFP) under handler, as well as the protocols package.

Behaviour: (Action)

None, one or more Behaviours **encapsulate actions** that cause interactions with other agents initiated by the AEA. Behaviours implement AEAs' pro-activeness. The framework provides ***abstract base classes*** that implement different types of behaviours e.g. cyclic/one-shot/finite-state-machine etc.

- **CyclicBehaviour:** If the agent is alive, this behaviour stays active and hence called repeatedly after every event.
- **TickerBehaviour:** A type of cyclic behaviour which periodically executes some user-defined piece of code.
- **OneShotBehaviour:** As the name suggests, it executes once and dies.

- **Finite State Machine (FSMBehaviour).** This is a computational model that can be used to simulate sequential logic i.e. to represent and control execution flow. This oversees scheduling the next state. **Fuzzy Logic** can also be combined with **FSM** to allow multiple states rather than single state and using probability to determine behaviour.
- Other behaviours are.
    - **CompositeBehaviour**
    - **SequenceBehaviour –** Executes sub-behaviour serially
    - **SimpleBehaviour:** Usually this basic behaviour class can be a more than sufficient substitute/better solution when other seemingly better Behaviours are found to have undesired quirks

We learn that actions/behaviours take place at different points in time and depending on the circumstances at the time they are taken, have certain effects. At any point in time, the future is determined by two factors: The **history, and current actions** of agents. For instance, the past alone does not determine where an agent delivers a set of information, that is determine by whether in fact it takes the appropriate action.

Models:

- none, one or more Models that inherit from the Model can be accessed via the **SkillContext**.
- **Task:** none, one or more Tasks encapsulate background work internal to the AEA.

Task differs from the other three in that it is not a part of skills, but Tasks are declared in or from skills if a packaging approach for AEA creation is used.

A skill can read (parts of) the state of the AEA and suggest action(s) to the AEA according to its specific logic. As such, more than one skill could exist per protocol, competing in suggesting to the AEA the best course of actions to take.

An example would be an **instance** where an AEA who is trading goods, could subscribe to more than one skill, where each skill corresponds to a different trading strategy. The skills could then read the preference and ownership state of the AEA, and independently suggest profitable transactions.

NB: A skill is quite impressive in the sense that ***more than one*** skill can exist per protocol, thus encouraging competition amongst each other in suggesting to the AEA the best course of actions to take. Hence skills are **horizontally arranged.**

The purpose of AEA is to negotiate and transact with each other, so having skills subscribing to different protocols allows them to dynamically decide the ownership state of an AEA and then independently suggest profitable transactions.

This flexibility stretches as far as the implementation, in the sense that a programmer can have a skill comprising of e.g.

- *if-then* statements OR
- deep learning or reinforcement learning agents

Naturally, the framework provides a default skill called **error**, but additional skills can be added as packages.

Main Loop

The main agent loop performs a series of activities while the **Agent** state is continuously being run.

- Act() : this function calls the **act() - *Action*** function of all active registered Behaviours. (**Remember that the *behaviours* encapsulate actions that cause interaction with other agents.** These actions are initiated by AEAs.
- React() :this function grabs all Envelopes waiting in the **InBox** queue and calls the **handle()** function for the Handlers currently registered against the protocol of the envelope. It is vital to note the **envelopes** should be processed **asynchronously**.
- Update() :this function dispatches the internal messages from the decision maker to the handler in the relevant skill.

Essentially the operation on an **agent** can be broken down into three parts.

1. Setup:
2. Operation:
    - Main Loop
    - Task loop
    - Decision maker loop
    - multiplexer
3. Teardown

The composition of an agent is seen by running a couple of dependencies within a CLI (Command Line Interface). After having prerequisites like **'Docker'** and **'Python'** installed. A Virtual environment is created to allow everything to be run holistically.

Simple steps as described by Fetch.Ai:

mkdir my_aea_projects/
cd my_aea_projects/
Download folders containing examples and scripts:

svn export https://github.com/fetchai/agents-aea.git/trunk/examples
svn export https://github.com/fetchai/agents-aea.git/trunk/scripts

Installation
pip install aea[all]

for *'**zsh**'* rather than *'**bash**'* type
pip install 'aea[all]'

An author name can then be setup using
aea init

An AEA can then be setup using
aea fetch fetchai/my_first_aea:0.5.0
cd my_first_aea

With the simple steps above we can view the basic components of an agent that is generated using the AEA framework as a modular system with different **Connections, contracts, protocols, and skills**. This is important since most AEA development focuses on developing one's own agent.

A deeper dive into the 'internal' features of an AEA

Protocols:

As viewed in the section above we obtain various modules when we created an agent. The **'protocols'** module that is created contains vital metadata within it, with this metadata, we can manipulate the code to determine governance role that our protocol will bear upon our agent.

The protocol thus manages message representation syntax in message.py, rules of the message exchange (semantics in dialogue.py), as well as encoding and decoding in serialization.py. All protocols are for **point to point interactions between two agents,** where an agent can be AEAs or other types of agent-like services.

Each message in the interaction protocol has a set of default fields. Which are.

- dialogue_reference: Tuple[str, str] , Note: the tuples are used as opposed to lists because tuples are immutable. I**f you have two agents, the first and second part of the tuple represent the message assigned to the first and second agents, respectively.**
- message_id: int ,Identifies the message in a dialogue. Its default value is **1**.
- target: int ,the id of the message being replied to. Its default value is **0**.
- Performative: Enum, the purpose/intention of the message
- is_incoming: bool ,this Boolean specifies if the message is outgoing, or incoming (from the other agent). The default Boolean is **False**.
- Counterparty: Address , the other agent participating in the transaction

Example, https://docs.fetch.ai/aea/thermometer-skills-step-by-step/

Skills:

A developer of AEAs writes skills that the framework can call. On creating an agent, a skills directory is created and as such includes the modules for the Behaviour, Task and Handler classes as well as a configuration file skill.yaml.

It is imperative to note that the three methods/modules within each class must have these two features implemented in each. i.e. **setup()** and **teardown()**.

Just like we spoke of Sharding earlier, skills are **horizontally layered**, meaning that they run independently of each other and thus cannot access each other's state.

Connections:

A Connection is attached to an AEA within the AEA framework. The connection.py module in the 'connections' directory contains a Connection class, which is a wrapper for an SDK or API. An AEA can interact with multiple connections simultaneously.

Configuration of a connection:

The connection.yaml file in the AEA directory contains protocol details, connection URL (Uniform Resource Locator) and port details. For example, the OEF connection.yaml contains the connection class name, supported protocols, and any connection configuration details. The developer is left to implement the methods of the Connection depending on the protocol type.

The AEAs structure can be perceived as a Merkel tree in blockchain where a series of immutable events operate in a verified, accurate, efficient, and quick manner.

- Adding new skills, protocols, and connections to the AEA.
- Scaffold generator

Here we use what is called a scaffold generator that builds out the directory structure required when adding new skills, protocols, and connections to the AEA.

aea create my_aea --author "fetchai"
cd my_aea

#Scaffold a skill

aea scaffold skill my_skill

Scaffold a protocol
aea scaffold protocol my_protocol

Scaffold a connection
aea scaffold connection my_connection

At this point one can develop their own skills, protocols, and connections. After then update the fingerprint and run the AEA

aea fingerprint [package_name]

https://docs.fetch.ai/aea/oef-ledger/

OEF (Open Economic Framework) or SOEF (Simple-OEF)

The OEF is an environment consisting of protocols, languages and market mechanisms that enable agents to search, find, communicate, and trade with each other. This as a result enables the creation of value from the inter-operability of OEF and DLTs (Distributed Ledger Technologies), and as it develops it will morph into a full-scale decentralized multi-dimensional digital world.

The Fetch.AI, OEF is functional because of two main components.

- A permission-less, public peer to peer (agent to agent) Agent Communication Network called the ACN (Agent communication Network) that allows agents to send and receive envelopes between each other, and
- A centralized search and discovery system. Peer to peer agents that are engaging in any form of transaction within the framework are mapped to each other through their cryptographic private and public addresses by using a distributed hash table.

How agents identify themselves.

Agents identify themselves in multiple ways, such as.

- Their valid **private and public addresses**
- **Assigned/given names** (names are not used to search for agents. The agents use them to identify themselves). Names could be Alice, Bob, flight, or train number etc.
- **Classification and genus.** This is the roughest description of what an agent is, and it is an easy way of filtering large groups of agents out of searches. Examples of genus categories are vehicle, avatar, buyer, IOT, data, building, services etc.

Brief description of how this all happens.

Agents register with the **soef** so that they can be able to search and discover useful services for their owners.

Workflow

- **Find** relevant agents on the soef,
- **Communicate** using the framework's peer-to-peer network
- **Negotiate** and transact on the ledger to exchange value tokens

Agents also describe themselves through several ways:

1. **Identify:** i.e. Ledger address, ledger type, given name. Ledger type can be Ethereum, Cosmos etc. Any distributed ledger that is interoperable with the fetch.ai ledger.

2. **Personality pieces:** How they look. The nature of an agent e.g. genus, classification, architecture, dynamics.moving, dynamics.heading, dynamics.position, action.buyer, action.seller

   Example:

   - **Genus** – vehicle, avatar, service, IoT, data, furniture(e.g. signs, mobile mast), building(e.g. railway station, school, hospital), buyer
   - **Architecture** – custom (custom agent architecture), agent framework (built using the agent framework)
   - **Classification** – mobility.railway.station, mobility.railway.train, mobility.road.taxi, infrastructure.road.sign

3. **Service keys:** What the agent does, sells or wants.

   Agents can have several service keys. Service keys hence are key/value pairs that describe the list of services that an agent provides. Just like personality pieces, service keys have no conventional format. e.g. **buying-genus, buying-architecture, buying-classifications**

4. **Range** (In Km)

All these are characteristics that an agent puts into consideration when searching and discovering to find the relevant data and service it needs. It is vital to have all these dynamics because the system must be in a state where it is very fast and efficient at executing tasks.

MAS – Multi Agent System

A multi-agent system is simply defined as a set of agents interacting with one another in a common environment to solve a common, coherent task. Agents that make up a MAS at times to achieve individual objects that conflict with each other, but their heterogeneity enables them to achieve their goals in parallel to one another without any hindrance and disruption.

MASs solve software problems by decomposing a system into multiple autonomous entities that are embedded in an environment to achieve the necessary task at hand. With Fetch.ai OEF, agents communicate with one another by sending messages that are uniquely identifiable within packages envelopes.

An agent may also be defined as an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments.

MASs have 6 learning aspects.

1) The degree of decentralisation
2) Interaction-specific Aspects i.e. what is the nature of the interactions among agents in the system? We can note that interactions amongst agents can change over time as they move through an environment.
3) We can have a local or global learning of the agents, hence the question. How involved are each of the individual agents in the learning process.

4) Goal-specific aspects: One must figure out whether the agents have a selfish or collectively set goal.
5) The learning algorithm. How the agents in the system learn.
6) The learning feedbacks

[5, pp. 259-298]

What type of machine learning seems more suited for MAS?

Reinforcement learning is the ideal approach as opposed to supervised, unsupervised learning. From the definition, "*reinforcement learning is an area of machine learning concerned with how software agents take actions in an environment in order to maximise the idea of cumulative reward"*.

Supervised learning creates a predetermined answer for what the agent is to expect and **unsupervised learning** would place the agent in a random learning environment which would not be ideal for corporative agent structure, not to mention that in both of these cases the learner would have to be supplied with training data.

Majority of research done in this field has been with the approach of reinforcement learning because it allows an autonomous agent that have no prior knowledge of a task, to learn behaviour by progressively improving its performance based on a rewards system.

How an agent's characteristics are built:

We have to note a fundamental philosophical dilemma when considering the creation of an agent, and it's as such: **"*An agent that constantly relies on a human for help is probably unhelpful, whilst an agent that never seeks assistance is potentially useless.*"**

Key behaviours that an agent should exhibit:

- autonomy: An agent encapsulating a state, that is not accessible to other agents, and can make decisions based on this state that are not affected by humans or other systems.
- reactivity: The agent can recognise changes within its environment (physical world, GUI, swarm of other agents, internet, or combination of environments) and then responding accordingly. The agent needs to have the capability of adjusting its strategy when executing a given task.
- Pro-activeness: The agent has the capability to work towards achieving a specific goal, whilst striking a balance between **reactivity** and **proactivity**. In other-words ability to exhibit goal directed behaviour.
- Social ability: The agent's ability to co-ordinate, co-operate, and negotiate with other software agents and possibly humans. All in with a purpose to achieve their goals in the best and most efficient ways possible.

Nothing better describes an agent than its ability to be **autonomous**. **Autonomy** essentially refers to an agent not depending on the properties or the states of other components for its functionality. An agent, unlike in Object oriented programming, should be able to make decisions depending on its **beliefs (modules),** hence the agent has sole control over the activation of its services and may refuse to perform a task. Thus, the agent's belief system is arbitrarily engraved within its **internal architecture.**

The internal architecture of agents, and how they react to a dynamic environment is highly dependent on the agent's **autonomy –** this architecture can be developed/built and represented by **abstract and concrete belief, desire, and intention** classes **(BDI)**, which essentially result into what we refer to as the **components of mental state.** An agent's purpose is essentially to reach a particular set goal by following a well laid out hierarchical plan to achieve said goal. An effective agent must have the ability to recognise and appropriately react to a current situation based on its belief system. Therefore, and agent must have the capability to understand its current state of existence from its target goal. [6, p. 7]

Decision making: Based on all the conditions brought forth, an agent's decisions are but logically constrained, though not determined by the agent's beliefs: these beliefs mostly refer to the **state of the world** (in the **past, present or future**), to the mental state of the other agents, and to the **capabilities** of this and other agents. For instance, if a robot believes that it cannot go through a wall, then it will decide not to smash through it. Decisions are also constrained by prior decisions, for example if an agent has decided to be at a particular location in 5 minutes, it cannot decide to be present at another location at the exact same time. [7, p. 11]

This also brings us to the notion that **Obligations should persist.** While an agent cannot unilaterally revoke obligations that it has towards others, it can cancel those that others hold towards it including the obligations that it holds towards itself. It is also imperative that the agent should also have the **capability** to fulfil an obligation, for it should know when to stop the pursuit of a task otherwise risk getting stuck in a loop.

The coding of an agent in the python language is dominated by mainly two aspects. (Class, and Object)

A class is created in the python language, under which there are various objects, each containing a set of attributes that define the rules/paths/commands that an agent will follow. These can be seen as the internal organs of each agent that allow it to operate as required in the environment in which they are placed.

We can see an example in the detection database of a car detector module

```
import logging
import os
import shutil
import sqlite3
import time

import skimage  # type: ignore
```

```
logger = logging.getLogger(
    "aea.packages.fetchai.skills.carpark_detection.detection_database")


class DetectionDatabase:
    """Communicate between the database and the python objects."""

    def __init__(self, temp_dir, create_if_not_present=True):
        """Initialise the Detection Database Communication class."""
        self.this_dir = os.path.dirname(__file__)
        self.temp_dir = temp_dir
```

We can also see a code snippet from a and AEA utilising the “gym.open.ai” to train a model

```
class GymHandler(Handler):
    """Gym handler."""

    SUPPORTED_PROTOCOL = GymMessage.protocol_id
    def __init__(self, **kwargs):
        """Initialize the handler."""
        nb_steps = kwargs.pop("nb_steps", DEFAULT_NB_STEPS)
        super().__init__(**kwargs)
        self.task = GymTask(self.context, nb_steps)
```

A well-developed Class and object architecture is used to give an agent its essence.

- Unpacking i.e. (**)
- __initi__
- super().__init__
- self.task
- self.context

What exactly is the agent environment?

The environments within which these agents reside are predefined by certain characteristics. As defined by Russel and Norvig, the environmental properties are classified as below; [3, p. 46]

- Accessible Vs inaccessible: Accessibility in an environment is where an agent can obtain complete, accurate, up-to-date information about the environment’s state. Environments as we know them today (i.e. the physical world and the virtual or the internet) are mostly not as accessible in this sense.
- Deterministic Vs non-deterministic: Deterministic refers to an environment where there are expected guaranteed outcomes given a certain action or set of actions,

such that there is no uncertainty. Conversantly a non-deterministic serve to the contrary.

- Static Vs dynamic: Static environment can be assumed to maintain its state or be unchanged by the agents existence within it, whilst a dynamic environment just as the physical world or the internet, has changes occurring within it that are caused by other operations that cause it to alter beyond the agent's control.
- Discrete Vs continuous: An environment is discrete when there are a fixed, finite number of actions within it. An either this or that environment with no outliers.

From these definitions we may hence deduce that the most complex type of environment is one that is **inaccessible, non-deterministic, dynamic, and continuous.**

Ideal scenarios of how an agent would react in each environment.

1) An agent operating in an environment should be able to understand the various nuances within that environment to have the ability to predict the future. If an agent can predict the future, it means that it can fairly perform its actions without giving preference to any one action. This concept is referred to as ***fairness*** *[Francez 1986]. The ability to make a prediction also ideally helps that agent understand the long-term consequences of the decisions that it makes.*
2) Agents are meant to have the incentive to negotiate amongst themselves to make the best possible decisions for the most ideal outcomes. To ensure that agents do not dwell so much on this process that they take more time that necessary to make an optimal decision, the concept of ***real time*** *interaction must be of paramount importance.*
    - *Agents should act within a specified time bound*
    - *Ideally a result must appear as quickly as possible*
    - *if an agent is meant to perform a repetitive task, then it must do this as quickly as possible.*

Whilst proceeding to putting all this into consideration, it is imperative to note that the essence of building an ideal agent with in an ideal environment would constitute that the agent strikes a balance between what is known ***goal-directed behaviour and reactive behaviour****. In other words, the agents has to be able to achieve its goal, let go of a goal, know when to terminate a given goal, all this dependent on the pre-existing environmental conditions that either positively or negatively influence the attainment of that goal. All this is determined by pre-set conditions such as time constraints, consequences as well as performing or stopping the specified action.*

Hence, based on the analysis of the dynamic nuances of the environment, each individual agent can determine its own behaviour by referencing Its goals and beliefs. We can also view an agent as an ontological entity that makes decisions based on the fundamental understanding of its **beliefs(type of ontology), desires (Goal/Objective), intentions (Plan) and Capability.**

Agent Oriented Programming(AOP)

AOP is a computational framework that in a way is a specialization of Object-Oriented Programming. We came to a realisation that agents tend to have characteristics/components such as beliefs, desires, intentions, capabilities etc., these components are also seen as the **mental state** of an agent that are made possible through the use of AOP.

AOP also introduces operators for obligation, decision, and capability. The agents are controlled by agent programs, that include primitives for communicating with other agents such as informing, requesting, offering, and in this paper, trading, negotiating etc.

An agent's individual behaviour is non-linear, and this is observed in various thresholds within the programming language such as with if-then rules and non-linear coupling.

Asynchronous behaviour is the key in the system being built. Implying that building agents would require looking at many factors such as memory dependent behaviour. i.e. some agents could have memory and others do not. A programmer would have to determine what kind of relationship to create between the two or amongst a multi agent system. However, it is maintained that no agent solely depends on another for it to function exclusively. This resulting individual behaviour brings in aspects such as learning and adaptation which can be driven by machine learning and artificial intelligence.

To the point of **heterogeneity**, agents are built to be diverse in character and content as mentioned above. This diversity can lead to network effects such as deviations from predicted aggregate behaviour. A heterogeneous environment allows the co-existence of software and hardware that have a structural makeup of internal objectives that may be fundamentally inconsistent and contradictory with those of others that reside within the same environment.

Asynchronization is the preferred method for designing agent processes and interactions. **Asynchronization** as opposed to **synchronisation**, enables an agent to act upon a request without necessarily having knowledge of that which has made the request. Synchronisation requires two agents to have prior knowledge of another's existence whereas **asynchronization** sees to it that agents exist independently in a given environment and only make decision such as accepting or denying requests through a process of negotiation.

The factors for designing an independent agent that has the ability to negotiate the most favourable terms are important because a requesting agent can provide:

- **Incorrect information**. An agent can lie about what it is willing to provide in return for the requested information.
- **Incomplete information**. Incomplete information may be traded.
- **Uncertain information**. Any of the agents may provide information that is as a result of poor simulation, poor training or any other such thing.
- **Detrimental information**. An agent can have pre-determined requirements from a requesting agent for information such as user identification to maintain privacy and security.

It is also imperative to note that this Agent Oriented Programming (AOP) style is fundamentally different from Object Oriented Programming style (OOP). The main difference is that **Objects** have **no choice** in deciding whether a service they provided is executed or not, for they just make it publicly available, whereas **Agents** tend to create a negotiation and the final decision lies with the agent receiving the request and not the one making the request.

The purpose of using agent programs is to control the evolution of an agent's **mental state;** The resulting actions occur as the consequences or side-effects of the agent's commitment (obligation) to an action whose time has come.

Summarised distinction between **AOP** and Object-Oriented Programming **OOP**

AOP	OOP
Generic role	Abstract class
Domain specific role	Class specific role
Knowledge, belief	Member variable
Capability	Method
Negotiation	Collaboration (uses)
Holonic agents	Composition (has)
Role multiplicity	Inheritance(is)
Domain specific role + individual knowledge	Polymorphism
Service matchmaking	

Typical applications of AOP

- Mobile computing
- Mobility
- Concurrent problem solving
- Proxy handling
- Communication traffic routing
- Information scouting

Understanding the historical evolution of computing puts into perspective that which is required to achieve a future that is governed by multi-agent systems. Everyone is scare of how computing might make the human brain obsolete, but multi agent systems seem to restore hope in the sense that they will be our guides to a deeper understanding of the parallels between the physical and the virtual environments.

Mike Wooldridge [8, p. 14] summarises this historical evolution in 5 terms.

- Ubiquity
- interconnection
- intelligence
- delegation
- human orientation

To top it off. In the case of **objects** in OOP, the decision to execute a given task lies with the object that **invokes the method**. In the case of **agents** in AOP, the decision lies with the agent that **receives the request**. i.e. **Objects do it for free, agents do it because they want to.**

Fundamental questions about the Agent and Multi-agent system.

Why are Agents and MASs important?

- Open systems. It may be difficult for a human to understand all the nuances pertaining the inner workings of an open distributed system, hence **MASs** foster an environment that can engage in flexible autonomous decision making that is critical for the development of complex systems.
- Legacy/outdated systems. The efficiency of older antiquated systems can be greatly influenced by increasing their reach and interaction with new computer technology. For instance, an autonomous agent layer that is fed with the operations of an older system can extend its functionality to one that is smarter and more advanced. For example, in transportation, older tracks can be fit with their own personalised autonomous agents that would foster communication with new automated autonomous vehicles.
- Data distribution or control. The combination of distributed ledger technology and MAS makes it possible for data stored in 'nodes' that are distributed in various physical locations to interact autonomously with each other. This is possible because the data is represented in the form of agents.
- Natural metaphor. MAS create an environment where there is trade happening 'naturally'. Arbitrarily an agent should be able to make its own informed decisions based on a self-governed belief system. Therefore, an agent would be able to devise the most effective methodology to employ when trying to achieve a particular goal such as making a decision around whether or not to execute a goods and services trade for its owner. The agent must pick the most favourable outcome for both parties to foster a short- and long-term relationship.

What are the fundamental architectural problems to look at when building an agent and consequently a MAS?

- **How do we go about building an agent?**

Design an agent that can makes its own decisions on how to perform specific tasks in the most effective way.

- **Society design.**

  How to design computer programs that know how to co-operate, co-ordinate and negotiate with one another.

The MAS architecture development process tends to focus on perspectives that are both the micro and macro paradigms that would make for a desirable agent. [8]

Conclusion:

Fetch protocol:

Fetch is a decentralised digital representation of a virtual world in which Autonomous Agents can reside and perform useful economic work on behalf of their owners. The interesting thing is that fetch is built with an immutable digital currency referred to as ETCH, that the agents reward themselves with after successfully completing a task. The agents are in-fact digital entities that can make educated and rational decisions on their own on behalf of their stakeholders. e.g. individuals, private enterprises, and governments. The Fetch ecosystem is facilitated by its unique smart ledger technology that leverages a consensus mechanism known as **useful proof-of-work** to facilitate high performance, low cost transactions. As a result, we get an environment that keeps building market intelligence and trust over time.

Fetch.AI's core proposition is to create a trusted environment in which Autonomous Economic Agents can exist, discover and be discovered, communicate with one another, broker and transact with the confidence that messages and transactions are both secure and between the intended parties.

A developer can leverage this environment to create agents of any calibre, purposed, use and intention.

The Fetch network runs on a decentralised node system where the Fetch node software is written in C++ for performance and stability. The software suit is built to minimise network traffic, maximize scalability, and be resource efficient.

This summarises the capabilities of the Fetch technology and briefly describes the programmability of Autonomous Economic Agents as far as their core epistemological features are concerned. Features such as goals, capabilities, beliefs, intentions.

Agents and agent based systems will have a major impact on the way we interact with technology and Artificial Intelligence. The fields of Reinforcement Learning, Generative Adversarial networks and Convolutional Neural Networks can greatly be greatly heightened when incorporated into agent based systems of learning.

Bibliography

[1] B. Geoghegan, “Agents of History: Autonomous agents and crypto-intelligence,” John Benjamins Publishing Company, USA, 2008.

[2] A. M. Mohamed and H. Abbas, “Multi Agents System for Industrial Applications,” Ahmed M. Mohamed, Egypt, 2013.

[3] S. J. Russell and P. Norvig, “Artificial Intelligence: A Modern Approach,” Prentice Hall, Englewood cliffs, New jersey, 1995.

[4] M. Wooldridge and N. Jennings, “Intelligent Agents: Theory and Practice,” The knowledge engineering review, Cambridge, 1995.

[5] H. weiss, Multiagent Systems: A Modern Approach to Distributed Modern Approach to Artificial Intelligence, 1st edition ed., G. Weiss, Ed., London, Cambridge, Massechusetts: The MIT Press, 1999.

[6] S. Park and V. Sugumaran, “Designing multi-agent systems: a framework and application,” Expert systems with Applications 28, Seoul, South Korea, 2005.

[7] Y. Shoham, “Agent-Oriented Programming,” elsevier Science Publishers B.V., CA, 1991.

[8] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd Edition ed., Liverpoool: John Wiley & Sons Ltd, 2009.

[9] D. Kinny and G. Michael, Modelling and Design of Multi-Agent Systems, Technical Note 59 ed., Melbourne: Austrlian Artificial Intelligence Institute, 1996.

[10] M. Wooldridge and P. Ciancarini, “Agent-Oriented Software Engineering: The State of the Art,” Liverpool, UK, 2000.

[11] T. Semwal and S. N. Bhaskaran, “AgPi: Agents on Raspberry Pi,” Electronics, Guwahati, 2016.

[12] L. Florin, “Design of a Multiagent system for solving search problems,” *Journal of Engineering Studies and Research*, vol. Volume 16, no. No.3, p. 14, 2010.

[13] K. M. Khalil, M. Abdelaziz and A.-B. M. Salem, “Machine Learning Algorithms for Multi-Agent Systems,” Cairo, Egypt, 2015.

[14] J. Kazil and D. Masad, “Mesa: An Agent-Based Modeling Framework,” Viginia, 2015.

[15] N. Criado and V. Botti, “Open issues for normative multi-agent systems,” gti-ia dsic, Valencia, 2011.

[16] Craig W. Reynolds, “Steering Behaviors For Autonomous Characters,” Sony Computer Entertainment America, Foster City, CA 94404, 1999.

[17] N. Hutton, J. Maloberti, S. Nickel, T. F. Ronnow, J. J. Ward and M. Weeks, “Design of a Scalable Distributed Ledger - Fetch.AI ledger Yellow paper,” Fetch.AI, Cambridge, 2018.

[18] M. Weeks, “The Evolution and Design of digital Economics-Fetch.AI Economics White paper,”

Cambridge, 2018.

[19] T. Simpson, H. Sheikh, T. Hain, T. ronnow and J. Ward, “FETCH: TECHNICAL INTRODUCTION. A decentralised digital world for the future economy,” An Outlier Venture, Cambridge, 2019.

[20] D. Galindo and J. Ward, “A Minimal Agency Scheme for Proof of Stake Consensus,” Fetch.AI, Cambridge, 2019.

[21] M. J. M.´, “Reinforcement Learning in the Multi-Robot Domain,” Kluwer Academic Publishers., Waltham, 1997.

[22] Fetch.AI, “Welcome to Fetch.ai developer resources. Let's get started,” Fetch.AI, 29 November 2019. [Online]. Available: https://docs.fetch.ai/. [Accessed 2020].