TRADING AGENT COMPETITION SPECIFICATION

v1

June 28, 2019

Fetch.AI

Marco Favorito
Ali Hosseini
David Minarsch

Contents

1 Introduction

This document is a specification for the Trading Agent Competition (TAC) framework. It accompanies the public release of our TAC python package, an open source implementation of the specification by Fetch.ai. The release of this framework also coincides with the first competition under the TAC framework organised by Fetch.ai.[1]

The purpose of this, and any future competitions under the TAC framework, is to foster a developer community around Fetch.ai's ecosystem, and to excite developers with the potentials of building on, and interacting with its various technologies, including the OEF, and later on, the ledger.

Our aim is to design competitions that resemble real applications of Fetch.ai's current or future technology. In other words, the scenarios designed for each competition will have strong ties with the experience of building on or using Fetch.ai's systems. As such, our hope is that through the competitions, participants will gain familiarity with using the various components of Fetch.ai's technology, and are introduced to its different use cases.

The vision for the first competition is to deliver a multi-agent competition that ignites excitement with respect to agent development. The competition is for anyone with an interest in creating agents on Fetch.ai's platform, including casual and experienced developers, researchers, and those interested in Fetch.ai's vision of a decentralised economy. No limit is placed on the number of participants and the type of teams. For the first competition, an in-person event, similar in form to a hackathon, will take place in which, as well as being able to compete, participants would

[1] There is a rich history of trading agent competitions, often under the TAC name (see [5]). This project has no direct affiliation with any of those projects.

have the chance to meet and engage with some of Fetch.ai's team members. For the future, our aim is to enable everyone to participate in the competitions fully remotely, with some competitions having an associated simultaneous in-person event for added engagement.

It has been of great importance for us to ensure the competition has a low barrier to entry while offering a rich domain with vast opportunities for innovative solutions. For instance, there will be a number of easily configurable, so called *baseline agents* provided by Fetch.ai, which could be fine-tuned with minimal modifications to get the desired behaviour, or extended with in-depth functionality.

The winner and runner-up teams in the competition will win FET tokens.[2] Note that the detail of the competition, including date and time, the competition's format, prizes, and the venue for the in-person event, will be specified in this link.

2 Competition

2.1 Target audience

The competition is designed with the following audience in mind:

- Individuals with minimal coding/development skills but with expertise or interest in some other area related to the trading agent competition (e.g. those with expertise in economics, finance, trading, multi-agent systems, agent strategy, etc).
- Casual developers interested in agent development.

[2] FET are utility tokens which can be used to deliver and consume services on Fetch.ai.

- Seasoned software engineers interested in agent development.

2.2 Scenario

The scenario of the competition is as follows:

> There are a number of different live performances in a festival that is coming up. Due to high demand and limited availability of space, the organisers of the festival have decided to use the following system for registration and ticketing.
>
> Every group of people who plan to attend the festival (e.g. families, groups of friends) must be represented by one computational autonomous agent who handles their registration and ticketing on their behalf. The agents should register the group of people they represent for the festival, at which point through lottery, each agent will be given a number of tickets at random to each performance.
>
> Of course each participant may have varying preferences towards the different shows, which may certainly be different to the other participants. Each agent is assumed to be aware of the preferences for the group of people they represent. However, crucially an agent does not know the preferences of other people as maintained by their agents. Once all the agents receive their tickets, they are given some time to gather the best combination of tickets for their own respective group, by trading tickets with other agents. The trades are bilateral (i.e. one-to-one) and using simulated FET tokens as the medium of exchange. At the end of this time period, agents are evaluated based on how

well they have satisfied the preferences of those they represent through trade.

As this is the first competition under the TAC framework, we decided to focus on a scenario that involves one of the most fundamental forms of economic interactions; bilateral trades. The competition involves a society of agents, each starting with a number of goods, in which agents could engage in one-to-one trades using simulated FET tokens as their medium of exchange.

Each agent, in the real world, would represent an individual or a group of people (e.g. a family) and is tasked with looking after their interest by maximising their welfare. In order to be able to do that, the agents would be made aware of their owners' preferences [4] and values [1]. In the competition, this is simulated by explicitly giving each agent a representation of their owners' preferences over goods at the beginning of each round. During the competition, the goal of each agent is to maximise its owners' interests by engaging in profitable trades, of course taking into account their preferences.

In the next section, the setup of the competition is more formally defined.

2.3 Setup

(**Agents**) There is a set A of agents partitioned into $A_{external}$ and $A_{baseline}$, i.e. $A = A_{external} \cup A_{baseline}$. Each agent $a \in A_{external}$ is provided by a competition participant. Each agent $a \in A_{baseline}$ is an instance of one of the baseline agents created by Fetch.ai. Note that $A_{baseline}$ can be empty, since it only serves the purpose of providing sufficient agent numbers during a competition, and for agents to train against during training. The game assumes a minimum number of agents min_agent (see Section 2.6 for

more details on this). Finally, there is a special agent c, called the *controller agent*, which runs the competition, and due to its special role, is not included in A. Section 3.3 describes this agent in more detail.

(Goods) There is a tuple of n sets of goods $X = \langle X_1, \ldots, X_n \rangle$ where each $i \in \{1, \ldots, n\}$ represents one type of good (e.g. festival ticket to a concert by Elton John) and each element of the set X_i is an instance of that good type (e.g. equivalent instances of festival tickets to an Elton John concert). In general, we may have $|X_i| \neq |X_j|$ for any two $i, j \in \{1, \ldots, n\}$ where $i \neq j$; that is, the number of available good instances could be different for two different goods. As a consequence, we may in general have differing aggregate supply across goods.

(Endowments) Agents are provided with *endowments*[3] in the goods. That is, each agent $a \in A$ has an endowment $e^a = \langle e^a_1, \ldots, e^a_n \rangle$, a tuple of length n, where the set $e^a_i \subseteq X_i$ is the endowment of agent a in good i. Each agent is given at least some $base\_amount > 0$ of each good, thus for any $i \in \{1, \ldots, n\}$, we have $|e^a_i| = \{base\_amount, \ldots, |X_i|\}$. We assume that endowments are allocated such that for each $i \in \{1, \ldots, n\}$ we have $\bigcup_{a \in A} e^a_i = X_i$; that is, all endowments for a good sum to the total instances of that good. Finally, every agent is endowed with the same $money\_amount > 0$, a real number.

(Current holdings) Each agent a's *current good holdings* are denoted by $x^a = \langle x^a_1, \ldots, x^a_n \rangle$ where for any $i \in \{1, \ldots, n\}$, $x^a_i \subseteq X_i$ is the set of instances of good i that agent a currently possesses. Therefore, $|x^a_i| \in \{0, \ldots, |X_i|\}$; note how agents can have no instance of a good i at some point in time but trivially never a negative amount. We further assume that for each $i \in \{1, \ldots, n\}$ we have $\bigcup_{a \in A} x^a_i = X_i$; that is, all agents' current good holdings

[3] An endowment is a subset of the good instances assigned initially to an agent.

of a given good sum to the total instances of that good available. Hence, that at the beginning of a round in the competition, before any trades take place, the good holding of any agent is equivalent to its endowment. Furthermore, each agent a's *current money holding* is denoted by m^a and it must always be the case that $\sum_{a \in A} m^a = |A| \times money\_amount$ and for each agent $m^a \geq 0$.

(**Preferences**) Agents are assigned *preferences* for goods and money by the controller agent c. That is, each agent $a \in A$ has a preference relation $\preccurlyeq_a$ on goods which totally ranks any possible combination of good bundles. Preferences are assumed to be transitive, thus for any three arbitrary goods x, y and z, $x \preccurlyeq_a y$ and $y \preccurlyeq_a z$ means $x \preccurlyeq_a z$. In practical terms, each agent a has a utility function u^a which is quasi-linear in goods and money:[4]

$$u^a(x^a, m^a) = m^a + g(x^a) = m^a + \sum_{i \in \{1,\ldots,n\}} s_i^a \times f(|x_i^a|) \tag{1}$$

such that $s_i^a > 0$ and

$$f(|x_i^a|) = \begin{cases} \ln(|x_i^a|) & \text{if } |x_i^a| > 0 \\ -1000 & \text{otherwise} \end{cases} \tag{2}$$

Breaking down the utility function in Equation 1, m^a is agent a's money holding, s_i^a is the utility parameter agent a assigns to good i, and $f(|x_i^a|)$ parameterizes the number of instances of good i that agent a has.

Figure 1 visualises the $f(|x_i^a|)$ component of the utility function (i.e. Equation 2). The figure shows decreasing returns; i.e. the agent values acquiring each additional instance of a good less than acquiring the previous instance.

An agent a's utility parameters for goods are represented by

[4] see `https://en.wikipedia.org/wiki/Quasilinear_utility`

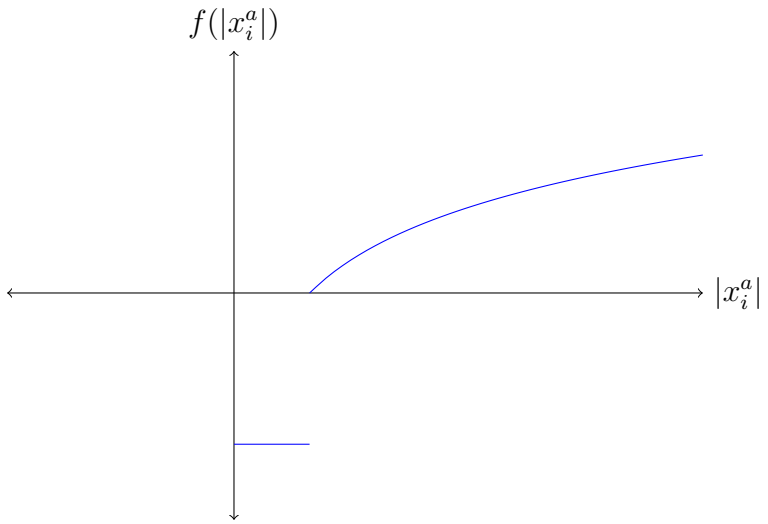

Figure 1: A visualisation of the $f(|x_i^a|)$ component of the utility function (i.e. Equation 2) for a good i held by an agent a. It shows decreasing returns; i.e. the agent values acquiring each additional instance of a good less than acquiring the previous instance.

$s^a = \langle s_1^a, \ldots, s_n^a \rangle$ where s_i^a is the utility parameter agent a assigns to good i. Equation 3 ensures the sum of the utility parameters for all goods are the same for every agent a:

$$\sum_{i \in \{1,\ldots,n\}} s_i^a = 1 \tag{3}$$

With the specific design of the utility function in Equation 1, we are implementing the well studied Cobb-Douglas function [2] for $g(x^a)$ which has the gross-substitutes property.[5,6] This property means that an increase in the price of one good causes agents to demand more of the other goods. In the interest of tractability and to guarantee an equilibrium, it limits preferences in that goods cannot be complements. Under certain assumptions, e.g. if transaction costs are zero, the property would also ensure that

[5] see `https://en.wikipedia.org/wiki/Gross_substitutes`

[6] Note, the Cobb-Douglas function has the same functional form as the negative cross entropy.

the game has a unique equilibrium.[7] Note however, that since we introduce a trade cost (see below), our game may have many equilibria.

In the context of the competition, the utility function serves as the metric which the agents have to maximise. The agent which best achieves this goal becomes the winner of the competition.

(**Trade cost**) We introduce a *trade cost* k to explicitly model transaction costs that would eventually occur on the ledger. This has the nice side effect that it limits against some forms of denial-of-service (DOS) attacks, provided the attacker cares about winning the competition. Each transaction incurs the same cost k.

2.4 Strategies

The focus of a rational agent will be to discover how they can arrive at an optimal bundle - as defined by their preferences - through successive trades. In the current specification, emphasis for agents is placed on a) finding the right agents to trade with, and b) doing the right trade with them. This might involve identifying other agents' needs to arrive at the optimal trading sequence. However, a dummy agent which simply selects random bargaining partners and trades if it is beneficial would still be able to improve its score. It would then be up to the developer to focus exclusively on simple message passing or extend the scope to include implementing elaborate discovery strategies by fully utilising the OEF, opponent modelling, or (multi-party) trading strategies. Crucially, we are not restricting complexity in agent development in this setup, rather we provide a point of focus, i.e. simple bilateral good exchange, for agent developers.

[7] This competitive equilibrium is achieved via a so called *Walrasian auctioneer* described here.

2.5 Discussion

The competition setup mirrors a *Walrasian Exchange Economy*, the workhorse model in economics for representing a market, which demonstrates the concept of *allocative/Pareto efficiency* well, i.e. allocating resources to those who will make best use of them.[8]

Since agents negotiate one-on-one, they should not be able to achieve the efficiency of the Walrasian Exchange Economy. In particular, it has been shown that in theory, an auction is more efficient than negotiation under most circumstances [3]. However, this does not mean that individual agents won't be able to outperform through negotiations relative to an auction-based market outcome. Specifically, due to the private information on preferences and good holdings, there is vast scope for some agents to perform better than others not by chance alone.

The reason for the above is that, from a game-theoretic perspective, the competition is not a zero-sum game. That is to say, an increase in one participant's score is not necessarily - and in general unlikely - at the cost of another. As a consequence, opposition to proposals and potential trade is weak in this setup since trades can be beneficial to all parties involved.

In this competition, agents' preferences and endowments are provided exogenously by the controller agent. Of course in a real-world use-case, the preferences and endowments would arise naturally. However, for the competition it is necessary to explicitly impose these on the agents to guarantee an interesting and fair setup.

Moreover, although the competition is designed so that agents

[8] An excellent introduction to Walrasian Exchange Economy could be found here.

would treat goods independently, for a hyper-rational agent, price changes in one good should lead to changes in demand for the other goods due to the gross-substitutes property. Hence, the aggregate supply in one good can affect the demand in another.

The current setup allows us to provide a competition with a relatively low barrier to entry and avoid a scenario where only experts in Artificial Intelligent (AI) or Multi-Agent Systems (MAS) could participate in the competition. Traditionally, MAS competitions targeted experienced teams of multi-agent developers. This has extensive academic and technological benefits, such as pushing the field forward. However it often leads to the exclusion of more casual developers from participating. Traditional competitions have a strong focus on optimal agent strategies and due to the environments agents are exposed to, this requires knowledge in advance optimisation techniques. Fetch.ai's aim is to deliver a competition where optimisation techniques are not strictly required for an agent to be able to compete.

The selected use-case is one of the most basic trading setups we could think of which enjoys a competitive element. Should it transpire that the setup is too simplistic (i.e. degenerate) then we will aim to extend it for future competitions.

2.6 Trading game phases

Pre-trading game phase:

- A trading game G consists of k game instances g, i.e. $G = \{g_1, \ldots, g_k\}$.
- Participating agents register with the controller agent during the $T_{registration}$ time period.
- If min_agent number (i.e. the minimum threshold) of partic-

ipating agents do not register by the end of the $T_{registration}$ period, then the controller agent will spawn a number of baseline agents $a \in A_{baseline}$ to meet the minimum threshold, i.e. until $|A| \leq min\_agent$. This is to ensure there are enough trading partners/options for each agent. Note that the min_agent parameter is tunable.

Each trading game instance g:

- A game instance lasts for $T_{instance}$ time period.
- A new draw of endowments and preferences for each participating agent is made by the controller agent.
- Agents are sent their endowments and preferences from the controller agent.
- Agents trade with each other, simultaneously buying and selling their goods:
- Trades need to be registered with the competition controller to be considered valid.
- After time $T_{instance}$ has passed, the league table of the trading game instance g is identified by the game controller.
- At the end of a game instance, a pause that lasts for T_{pause} is taken.

Post game phase:

- The controller agent will report the final league table by computing a (weighted) average across all game instance league tables.

Altogether, the competition lasts for: $T_{registration} + [k \times T_{instance}] + [(k-1) \times T_{pause}]$.

3 TAC Components

3.1 Code Development Objectives

When we set out to develop TAC, we determined a number of design objectives:

- Developers should be able to extend baseline agents.
- Developers should be able to easily swap out components of baseline agents.
- Competition organisers should be able to easily adjust the game configuration.
- Agents should be able to make all decisions autonomously.
- The controller should mirror a smart contract and eventually be implemented in a smart contract on our ledger.
- References in agents, communications, and objects should be as explicit as possible, e.g. to any data models, ontologies, protocols and languages used.
- The agent architecture should logically separate the agent's strategy (i.e. its decision making component), protocol adherence (i.e. units that ensure the agent's behaviour complies with protocols) and beliefs (i.e. the agent's internal belief about the the state of the world).
- Agents should be proactive and reactive.

A key focus for us in the development of the python package is to ensure that both the competition framework and the agent framework allow for high modularity and composability. As we move towards more sophisticated architectures, we will endeavour to maintain those requirements.

The community is invited and encouraged to contribute to our framework development.

3.2 Network Environment

The competition will be held in a common identity managed environment. Fetch.ai will host the environment, consisting of *OEF node*(s), a competition manager – implemented via a controller agent – and a set of baseline agents, and will make it accessible via the web. A KYC provider will vet each participant which will allow us to provide environment access credentials to the competition participants. As Fetch.ai is moving towards deploying its ledger main-net, we will decentralise more and more aspects of the competition management, with the vision that eventually anyone could implement their own agent competition subject to their own specification.

The OEF nodes come with the *OEF* functionality. One such functionality is message relaying. This means that through appropriate APIs, agents could send and receive messages to each other, and the OEF will handle the low level delivery of the messages over the underlying network. Another *OEF* functionality is service registration and service search. Agents can register services on the *OEF* and search for services registered on it.

The environment will also be made available as a sandbox on a Docker image. The sandbox will be modular and configurable. The host of the sandbox will be able to configure the competition metrics (e.g. endowment and preference generators, $T_{registration}$, $T_{instance}$, T_{pause}). A sandbox can then be run a) without adding any agents, in which case a number of baseline agents will compete with one another, b) with a mix of baseline agents and custom agents from the developer, or c) with no baseline agents and only

agents from the developer. The agents provided by the developer connect to the sandbox via a port.

3.3 Controller

Fetch.ai is progressing towards launching a distributed ledger system.[9] The ledger is a secure, decentralised mechanism for storing and keeping track of information, such as the transaction history of agents in the competition. Furthermore, the smart contract functionality of the ledger, amongst other things, could be used to run and manage a competition.

Due to the ledger's absence in the current competition, a special controller agent c takes on the dual responsibilities of transaction settlement and competition management. This effectively means that the controller agent is the only entity that will hold the global state of the competition. Therefore, all participating agents must register their transactions with the controller agent for them to be considered valid.

Furthermore, the controller is responsible for generating the endowments and preferences of all agent at the beginning of each game instance, ensuring that they are somewhat orthogonal. This is implemented through a *generator* algorithm in the game controller which acts as the random seed to each game instance.

To summarise, the controller agent's responsibilities are as follows:

- At the beginning of each game instance, the controller agent generates and assigns the endowments e^a and preferences s^a of every participating agent a.

[9] Details of this can be found on www.fetch.ai.

- During each round, the controller agent registers and settles transactions, ensuring the feasibility constraints listed in *current holdings* in Section 2.3 are met. This means that the controller agent essentially keeps track of the good and money holdings of every participating agent during the course of the competition.
- At the end of the competition, the controller agent constructs a league table containing the final scores and ranking of all participating agents.

3.4 Baseline agents

3.4.1 Reactiveness vs Proactiveness

When designing an agent, it is important to pay close attention to the environment in which the agent will eventually be deployed. This is because the nature of the environment often imposes certain requirements on the characteristics of the agents which are to 'live' in the environment while successfully meeting their design objectives.

Of such characteristics, two of the most fundamental ones are *reactiveness* and *proactiveness*. The first refers to an agent's ability to respond to changes in the environment. For example, an agent responsible for controlling the central heating/cooling system of a house has to be capable of reacting to changes in the temperature of the house, and act accordingly.

On the other hand, proactiveness refers to an agent's goal-oriented tendencies. A proactive agent always has some state(s) of the environment in mind (i.e. its goals) which it tries to achieve through a, perhaps complex, series of actions (i.e. a plan), either individually or by cooperating with others.

Note that it is not difficult to create purely proactive agents. Such an agent would fixate on a goal state, and if the pre-conditions of its plan of action are met, executes its plan, after which it expects to have achieved the goal state, in other words satisfying the post-conditions of its plan. In systems where the environment does not change, a purely proactive agent may be sufficient. However, in environments which are dynamic, i.e. where the environment constantly changes, multi-agent, whereby other agents also make changes to the environment, or complex/uncertain, where it is difficult for the agent to fully observe the environment, a purely proactive agent would fail. For instance, when the preconditions of a plan suddenly becomes false while the plan is being executed, or when a goal state, i.e. the reason for executing the plan, is not valid anymore, then there would simply be no reason to continue executing the plan.

It is equally easy, and to a similar extent ineffective, to create purely reactive agents that just respond to the environment. What is desirable is striking an effective balance between proactiveness and reactiveness; agents that move systematically towards achieving their goals, while constantly evaluating their goals and plans, responding to any change in the environment that affects them, in time for their reaction to be of value.

Typically in multi-agent systems, incoming events that an agent has to respond to take the form of messages that it receive from other agents in the environment. These messages might range from requests for specific resources the agent offers, to broadcasts that a specific public contract was created, to regulator agents informing the agent that its actions may have breached the environment's protocol.

The baseline agents provided by Fetch.ai are designed to exhibit goal-oriented behaviour while having the capability to react to

changes in the environment. From a technical perspective, the agents have a so called *main loop* and an *event loop*. The former controls the agent's proactive behaviour, where at each 'tick' of the loop, the agent moves towards achieving its goal. The event loop on the other hand is responsible for processing incoming events. In our implementation, incoming events (i.e. messages) are placed in a queue by the event loop, which can then be processed in the main loop.

More specifically, the main loop in a baseline agent ticks every (say) 10^{th} of a second, where each time through the loop it:

1. Processes any incoming messages.
2. Updates the agent's internal state.
3. Allows the agent to make decisions and act.
4. Waits until it is time to "tick" again.
5. Goes to 1.

The key is to ensure that any message handling and internal state update happens fast enough to always (or mostly) avoid going over the allowed tick duration (e.g. 10^{th} of a second). The benefit would then be that the whole process could be kept single threaded and the code straightforward to read. If there is a need for some heavy processing, then there are a few options:

- Splitting the task up into chunks and processing them over several ticks (storing any intermediate internal state).
- Making the tick duration longer.
- Using language specific features (e.g. co-routines in python).
- Splitting the process across different threads, having the main thread just wait until all threads are finished with their job.

3.4.2 Baseline Agent with no world model

This is the most basic baseline agent developed by Fetch.ai that is able to compete.

Recall that each agent a has a current good holding $x^a = \langle x_1^a, \ldots, x_n^a \rangle$, where each $i \in \{1, \ldots, n\}$ is a good type and each x_i^a is a set of instances of good i that agent a currently has. Furthermore, each agent a has an associated utility function $u^a(x^a, m^a)$, which given a good holding x^a and money holding m^a, produces a number representing "how good agent a's current situation is".

The utility number could be used by the agent to compare different states and subsequently help in decision making. For instance, consider the situation where agent a has the good holding $\langle x_1^a, \ldots, x_n^a \rangle$ and is considering how much better off it would be for him to acquire another copy of good 1. This problem could be equivalently represented as comparing the utility values u^a of the two holdings $\langle x_1^a, \ldots, x_n^a \rangle$ and $\langle \tilde{x}_1^a, \ldots, x_n^a \rangle$ where $|\tilde{x}_1^a| = |x_1^a| + 1$.

In general, for an agent a to compare two states with good and money holdings of respectively x^a, m^a and $\tilde{x}^a, \tilde{m}^a$, the agent has to calculate the *marginal utility* $u^a(\tilde{x}^a, \tilde{m}^a)$ - $u^a(x^a, m^a) = g(\tilde{x}^a) - g(x^a) + \tilde{m}^a - m^a$. This is the relevant quantity for the agent to assess whether an exchange which results in its good holdings to change from x^a to $\tilde{x}^a$ and its money holdings to change from m^a to $\tilde{m}^a$ pays off. In particular, for the exchange to pay off, the change in the marginal utility has to be positive.

A baseline agent without a world model $a \in A_{baseline}$ will trade as follows. For each good i in her current holdings $\langle x_1^a, \ldots, x_n^a \rangle$, the agent will offer to sell one instance if she has at least two instances of the good and, simultaneously, the agent will offer to buy one instance. The *asking price* p is *equal* to the agent's marginal utility for i in the goods component only: $p = g(x^a) - g(\tilde{x}^a)$,

where $\tilde{x}^a$ is the current good holding x^a minus the instance of the good to be sold. Thus, the agent accepts any proposal for it to sell an instance of good i at some price p' *equal to or more than* p. Similarly, the *offer price* p is *equal* to the agent's marginal utility for i in the goods component only: $p = g(\tilde{x}^a) - g(x^a)$, where $\tilde{x}^a$ is the current good holding x^a plus the instance of the good to be bought. Thus, the agent accepts any proposal for it to buy an instance of good i at some price p' *equal to or less than* p.

With the above trading strategy, the agent will not make any profit in negotiations where it proposes the price. It can only make a profit in negotiations where is receives a proposal.

In the following sections we describe the negotiation as well as search and discovery process implemented by the baseline agent.

The registration and search process of services works as follows. An agent first generates a description of the services (i.e. goods) it offers according to the OEF data models. It then registers the service on the OEF. An agent can search for the services of all agents who are registered on the OEF. The agent does so by formulating a query in the OEF query language and sending this query to the OEF. Once the service search request has been submitted to the OEF, the OEF processes it and then returns a list of agents which match the query to the requesting agent. This process is visualised in Figure 2.

The negotiation protocol loosely follows the FIPA specification. This means, there is a multi-step dialogue during which agents negotiate. This is illustrated visually in Figure 3. The agent who initiates a negotiation by sending a CFP (call-for-proposal) to the counter-party. The counter-party, if it processes the CFP according to FIPA, then reacts with either a Propose or a Decline. A Decline ends the negotiation (see Figure 4). A Propose contains a list of proposals, which ideally match the query delivered in the

Figure 2: A sequence diagram of a service registration and service search.

CFP. Once Agent 1 receives the Proposal, and if it follows FIPA, it will either Accept or Decline it. If it declines, the negotiation ends (see Figure 5). If it accepts then it will also send a message confirming the transaction to the controller. The negotiation is successfully completed if the counter-party follows suit with a matching Accept and a transaction submission to the controller.

The second Accept is what we refer to as a *Match Accept*. Without this stage, Agent 1 would have to send the transaction to the controller at the propose stage. As the propose stage has a high probability of leading to breakdown in negotiations this would constrain the agent too much. This is because any transaction submitted to the controller must be considered in the *forward looking* state of the agent, i.e. the state the agent finds itself in once all the trades it committed to have been settled by the controller agent c.

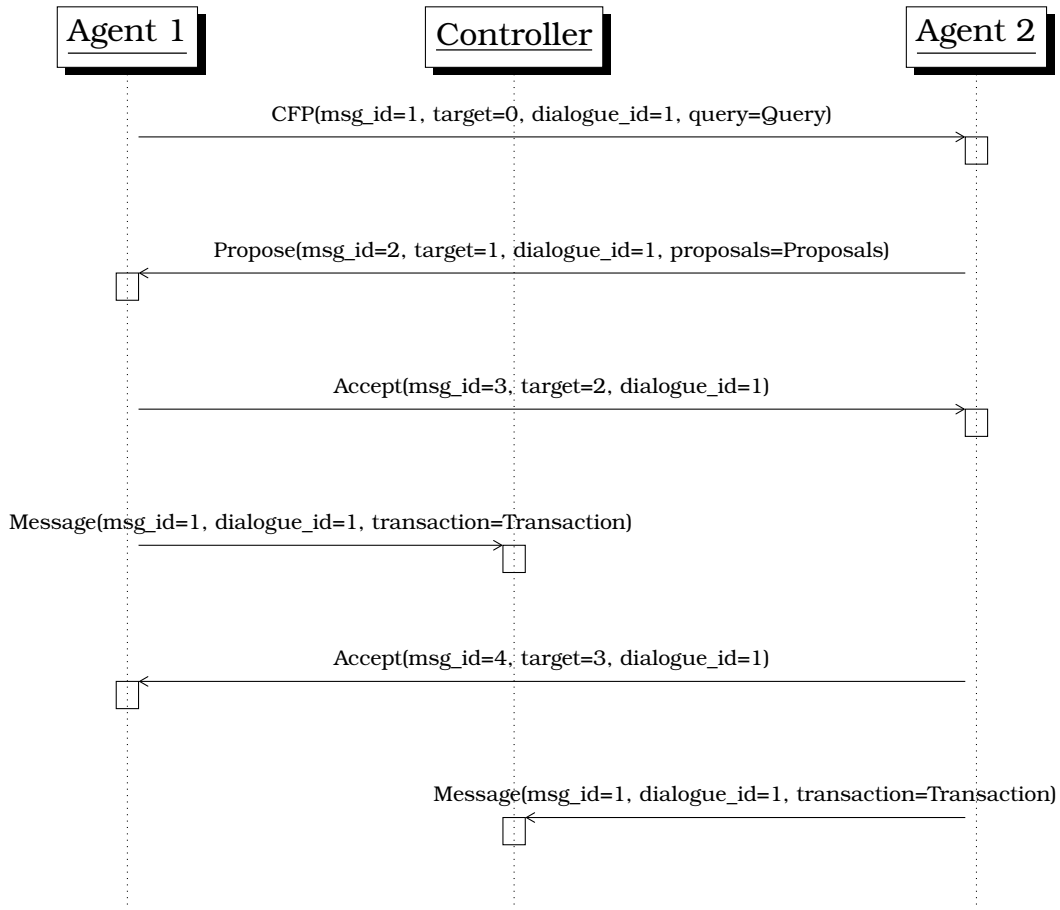

Figure 3: A sequence diagram of a successful negotiation ending with settlement of the transaction via the controller.

3.4.3 Baseline Agent with world model

The baseline agent with world model uses information gained from acceptances and declines of the proposals she makes to create a price model for each good. It then uses the price model to offer the price which it assumes has the highest likelihood of being a successful trade.

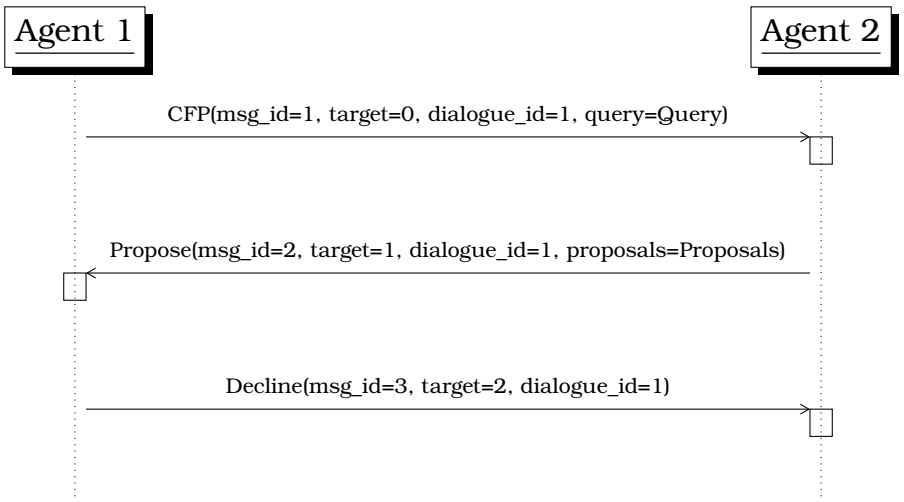

Figure 4: A sequence diagram of a negotiation which breaks down on the Propose because Agent1 does not deem the proposal beneficial.

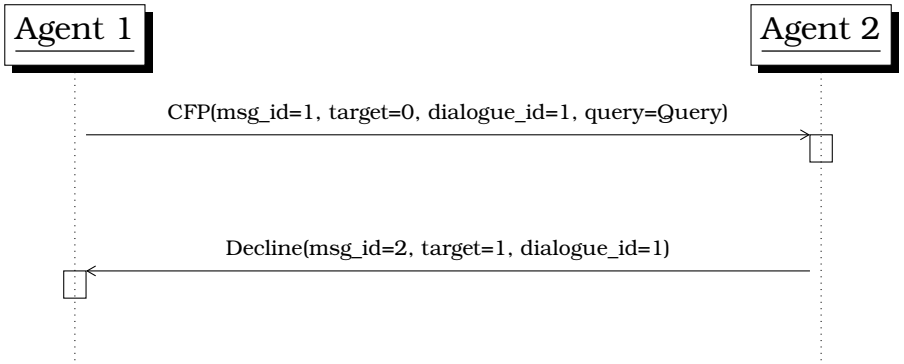

Figure 5: A sequence diagram of a negotiation which breaks down on the CFP because Agent2 does not have anything to propose based on the CFP query.

3.5 Controller GUI

We provide a GUI for competition organisers to monitor the competition in real time and to help with analysing historic competition data.

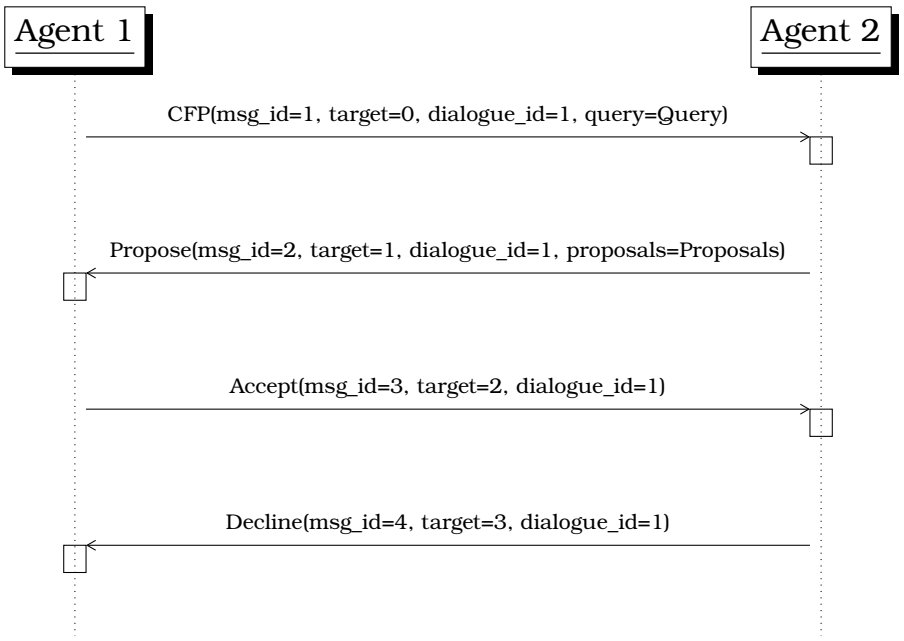

Figure 6: A sequence diagram of a negotiation which breaks down on the first Accept because Agent2 does not deem the proposal beneficial anymore (e.g. its trading position has changed).

3.6 Agent GUI

We provide a GUI for agent developers to monitor their agent in real time and to help with analysing historic agent behaviour.

4 Developer Community

4.1 Communication

A dedicated communication channel is opened for agent developers to exchange ideas and communicate with Fetch.ai developers. More details could be found in https://hackathon.fetch.ai.

4.2 Code Sharing

Developers who enter the competition will likely want to improve their agent. For this reason - and in the spirit of openness - we encourage and incentivise participants to share their agent code. However, it has been decided that no participant is forced to share their code in order to participate. For some competitions, we may require participants to describe the approach they took in developing their agent if they want to claim their reward.

5 Closing Remarks

5.1 Rules

The following regulations are enforced in the competition:

- Only one agent is allowed per individual/team.
- Any action that goes against the spirit of the competition is strongly discouraged. Examples of such behaviour includes, but is not limited to:
  - Launching any form of attack on any of the components of the competition, such as the OEF, other agents, the competition manager agent.
  - Participants' collusion with one another, through sacrificing one agent's score to increase another agent's score.
- Every individual/team wishing to participate in the competition has to go through a KYC procedure to claim their prize. The details of this could be found in www.fetch.ai.

- Agents can in principle communicate peer to peer. This is a realistic expectation which is difficult to enforce against.

5.2 Future TACs

There are many ways the trading environment could be made richer, and the domain more complex. A natural extension is to introduce a market through a centralised auction process. Theoretically, this should cause the agent-to-agent negotiation to unravel (for more information, please see here). However, it would be interesting to observe what happens in a multi-agent world where agents are unlikely (programmed to be) hyper-rational.

Below we list a number of features which can enrich a competition:

- **Richer strategies**: agents to be required to deploy strategies based on a variety of techniques (e.g. RL, other ML techniques, Evolutionary/genetic algorithms, logic-based, etc).
- **Multiplicity of issues**: so several agent skills are needed and no single type of agent strategy is superior in all markets.

When developing competitions we aim to avoid arbitrary elements which do not mirror reality. For example, involving a combinatorial auction in a setting where there is no evidence of such an auction being conducted in that market in reality.

References

[1] K. Atkinson and T. Bench-Capon. States, goals and values: Revisiting practical reasoning. *Argument & Computation*, 7(2 - 3):135 – 154, November 2016.

[2] M. Brown. *The New Palgrave Dictionary of Economics*. Palgrave Macmillan UK, 2017.

[3] J. Bulow and P. Klemperer. Auctions versus negotiations. *The American Economic Review*, 86(1):180–194, 1996.

[4] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton Classic Editions. Princeton University Press, 1944.

[5] M. P. Wellman, A. Greenwald, and P. Stone. *Autonomous Bidding Agents, Strategies and Lessons from the Trading Agent Competition*. Intelligent Robotics and Autonomous Agents series. The MIT Press, 2007.