Coordination in Utility Managed Multi-Agent Groups

Fernanda Barbosa¹, José C. Cunha¹, Omer F. Rana², Steven J. Lynden²
(1) Departamento de Informática, Universidade Nova de Lisboa, Portugal
{fb,jec}@di.fct.unl.pt
(2) Department of Computer Science, Cardiff University, UK
{o.f.rana,s.j.lynden}@cs.cardiff.ac.uk

Abstract

A two stage approach to co-ordination in a multi-agent society is presented. The first stage involves agents learning to co-ordinate their activities based on local and global utility – the latter being assigned by an Environment Service Node. Each agent aims to maximise its local utility over a single-step or finite horizon, subject to reinforcements being received from the ESN. An agent may reveal all or partial state to the ESN, and does so through specialised function operators. Groups are formed by agents trying to identify partners which enable them to achieve a higher local utility. Subsequent interactions within the group are structured and managed via GroupLog – an approach for supporting interactions between agents within a community. An example based on resource sharing within a Computational Grid is used to illustrate how the two approaches may be combined.

1 Introduction and Related Work

As the complexity of distributed systems has increased, de-centralised co-ordination has become an important goal. The emerging trend away from Client/Server-based approaches to Peer-2-Peer (P2P) systems, indicates the need for providing co-ordination techniques which are more dynamic, can scale, and do not necessitate a fixed structure. Generally, in P2P systems, it is the grouping of peers that is important, rather than the activities that must be undertaken by each peer. A good survey of concerns can be found in [8]. Establishing pre-defined co-ordination strategies may not be appropriate in these systems, and the “cookbook” approach applied in existing systems may be of limited use. Rasmussen [10] provides a survey of different co-ordination approaches in the context of multi-agent systems, and classifies these into a number of coarse-grained categories: (1) Game Theoretic, (2) Market-Oriented Programming based approaches, (3) Contract and Commitment based approaches, and (4) those based on Social Conventions. These categories employ a number of different themes, such as planning/reasoning, utility, and learning (for instance), to support co-ordination. It is also possible to combine themes, for instance, having social conventions (contracts) to constrain the behaviour of selfish agents.
An ‘agent’ in this work is used to identify a resource provider or a resource user, where interaction between the two are based on a policy.

The WALRAS algorithm [11] provides one approach for co-ordination based on utility, although this is in the context of very specific constraints on how the utility function (monotonicity, for instance) is calculated and assigned. This research, closely related to the first theme in our work, is based on the assumption that each agent owns resources, and can trade these in a market-place. Interaction does not take place directly between the agents, but via resource “prices” managed by them. There is no aspect of commitment in this approach, as agents can dynamically change their behaviours (and hence interaction patterns with others), based on prices for particular resources in the market. Although a useful and important paradigm within a dynamic environment where agents may have conflicting local goals, this approach may not be able to guarantee useful behaviours if commitments made by agents change often (as the market changes). In this approach, it is also not possible to guarantee that agents will continue to participate in a given market over a particular period of time (this may be important in the context of supply-chains, for instance, where agents need to commit for a particular period of time).

We use a two layer approach for managing agents the first is based on a utility model to co-ordinate agents, and create agent communities. This model assumes the agents are non-cooperative, and are interested in maximising their local utilities. The incentives for agents to create groups is therefore based on them maximising their own local utilities through the collaboration with their group peers. The second layer is then based on managing this group, once it has been formed. The duration over which participants of a group can remain stable is determined by the dynamicity of the environment in which they operate (although we do not address this issue in this paper). This layer assumes that agents within a group trust and cooperate with other members of a group and that an entire group may be viewed as a single entity. The approach provided is also a demonstration of how two different styles of coordination (and implementation) may be combined to yield a system that can support: (1) both selfish and cooperative agents, (2) selective disclosure of internal state by combining tuple space ideas [5] and messaging, and (3) the concept of time horizons to maintain group structure. Hence, initially agents do not commit to any particular actions, and operate in a selfish manner. They may subscribe to a number of different communities and based on the utility they receive from these communities, decide to stay or leave. The second phase involves some commitment from the agents to continue to work collectively over a particular period of time. This is controlled by the concept of a “time horizon” in our approach.

2 Coordination in Dynamic Communities

Coordination within a community of agents in our approach is achieved through a two phase process. The first phase involves agents learning to form mutually
beneficial communities, managed by Environment Service Nodes (ESNs). Each ESN is responsible for validating the usefulness of an agent to a given community, and achieves this by allocating a utility to each participant within the community that it manages. In the first phase, agents may subscribe to a number of concurrent communities (and hence ESNs), and get allocated utility from each one of these ESNs. Utility is based on two related aspects of an agent: (1) functional utility - a utility value an agent receives based on its problem solving capabilities, and (2) performance utility - a utility value based on the resource usage of an agent. It is important to note that utility is only useful if the source of the activity that leads to high utility (high reward) is deterministic (given a stationary environment). Hence, an underlying assumption is that agents will be able to determine the source of high/low utility. Each agent aggregates utility received from different ESNs and uses this to decide whether or not it should remain within a particular community. In the first phase, agents therefore learn to choose between the available communities and do not reveal their internal properties to other agents within their community. Direct interaction between agents is possible, and achieved through FIPA-ACL messages. There is no shared state within the community. Coordination between agents in this phase is regulated through utility assigned via the ESN. At the end of this phase, a data structure containing: (1) the list of agents present within a community, (2) the utility value allocated to each agent, (3) the group utility as calculated by the ESN, (4) the service types managed by each agent, and (5) the agent types, are passed on to the next phase.

The second phase involves agents being more cooperative, and revealing their internal properties through a shared group state, based on the Linda model [5], or via a direct one-to-one interaction. The shared space is a data structure that is made available to all participants within a community (i.e. a group managed by an ESN). The data structures contain attributes obtained from the first phase, and contain properties of each agent (such as its identity, its utility, etc) and the group state (such as group utility). The shared group state is a multi-set of atoms, defined in a module by the predefined interface predicate “state”. The members of the group may interact by accessing the shared group state, i.e. by reading and writing state elements. The predefined Linda-like predicates - \( \text{rd}, \text{ts}, \text{in} \) and \( \text{out} \) allow access to the state of a group. In this instance, the group hides the identity of its members from the outside, but allows redirection of messages to them via group interface predicates. The following notation is used to define participants of a community:

1. The set \( A = \{ A_1, ..., A^a \} \) of agents.
2. One or more Environment Service Nodes (ESNs).
3. FIPA platform agents: the directory facilitator agent (DF) and the agent management service (AMS).
4. The set \( T_{up} = \{ T_{up_1}, ..., T_{up^n} \} \) of tuples, where each tuple represents some property of a given agent community.
The DF and AMS services need to be shared by participants, and discovery is supported by mapping agent properties to addressing information managed by the DF. The following definitions are made:

Agent Service Types A set of service types \( T = \{ T^1, \ldots, T^m \} \) are defined for all available groups. A valid agent must implement a service type for instance, leaf.cg.data may define a specific type of agent managing data. Agent type definitions are globally consistent, allowing other agents, and the ESN, to make assumptions about an agent’s capabilities and behaviour once the agent’s service type is known. Agents must register their service type with the local platform DF (to be FIPA compliant).

Observable Property Set For each service type \( T^i \), an observable property set \( O^i \) is defined as a set of dynamic properties belonging to agents implementing service type \( T^i \). Each property has a globally unique name and a value associated with it. \( O^i \) is observable in the sense that an agent of type \( T^i \) can be required to inform the ESN of the values of each property in \( O^i \). These properties are primarily used in the first phase whereby agents may restrict access to all their internal properties to the ESN. In the second phase, agents may also restrict access to their internal state to the observable property set. Hence, the total state of an agent consists of \( O^i \cup L^i \) where \( L^i \) represents the non-observable, local properties.

Global Utility Function The global utility function, \( U_g \), rates the overall performance of a peer community.

Local Utility Functions Local utility functions are assigned by the ESN to agents. The local utility function assigned to agent \( A^i \) is referred to by \( U^i_1 \).

2.1 Utility Assignment

Utility assignment relies on the assumption that agents are trying to maximise some local metric and often work with uncertain knowledge about other agents. This is different from plan based systems, where agents must interact with each other to fulfill a given overall goal. Utility assignment and management is achieved using the LEAF toolkit [7]. Coordination in LEAF is based on the utility function assignment techniques of COIN [12], and support for this is incorporated into the LEAF infrastructure and builds on FIPA-OS [9]. The essential concept is that machine learning based agents can learn to maximise their personal local utility functions, which are assigned to agents with the aim of engineering a system in which improvements made to local utility are beneficial to the system globally. This concept is inherited from the COIN framework and require agents to undertake local optimisations based on a reward they receive for their operations. Global utility is therefore based on the observable property sets of the agents that comprise the agent group.

The set of observable properties required by the ESN from each agent implementing service type \( T^i \) is defined as \( O^i_g \subseteq O^i \), where \( O^i_g \) may be empty if
no parameters are required from agents with a particular service type. In many cases, multiple instances will exist of certain service types, meaning that the global utility function must be able to handle parameters with multiple values. On start-up, the ESN distributes local utility functions to all agents. For every agent $A_i$, its local utility function $U_i^j$ will depend on a number of parameters. Each parameter of $U_i^j$ is either (a) a member of $O_i$, or (b) must be supplied by the ESN. Parameters falling into category (b) for agent $A_i$ are defined by the set $R_i$. If the ESN assigns the utility function $U_i^j$, it is then responsible for dynamically updating the parameters $R_i$. Utility may be calculated by an agent based on each action that it takes, or averaged over a sum of states. Hence, an agent may use one of the following models:

- **Finite Horizon** optimise expected reward for next ‘$h$’ state updates:

$$E(\sum_{t=0}^h U^t)$$

using this update scheme, the agent group exists for only ‘$h$’ states, and is then disbanded. Hence, over its lifetime, an agent may belong to different groups (managed by different ESNs).

- **Infinite Horizon** long-run reward expected by the agent. In this case, the agent may not get an initially high utility, but expect to improve utility over a given number of time/states:

$$E(\sum_{t=0}^\infty \gamma^t U^t)$$

where $\gamma^t$ discounting rate for future states. Using this scheme, the agent group remains intact forever and utility update is primarily within each group by the ESN.

- **Averaged Finite Horizon** (gain optimal policy)

$$\lim_{h\to\infty} E(\frac{1}{h} \sum_{t=0}^h U^t)$$

this is midway between an Infinite Horizon and a Finite Horizon scheme whereby the further away the reward, the less important it is to an agent. Using this, an agent may be able to optimise its utility over the “medium” term.

Hence, the kind of update mechanism in place is influenced by the rate of change of the environment in which the agent exists. This can also be determined by a user and we assume an infinite horizon at present (section 3 discusses
the impact of this parameter with reference to a particular example). This parameter also influences interaction between phase 1 (managed by LEAF) and phase 2 (managed by GroupLog). Local and global utility functions are designed with default parameters, so that the functions can still be computed if some parameters are unavailable. For the first phase, interactions are as follows:

1. The ESN must locate agents, and determine their service types via the FIPA service descriptions which agents are required to specify when they register with the platform DF agent.

2. The ESN must inform all agents of each type $T^i$ of the observable properties $O^i_j$ that it needs to compute $U_j$.

3. Each agent of type $T^i$ must dynamically inform the ESN of the values of the observable properties $O^i_j$.

4. The ESN must dynamically inform each agent $A^i$ of the values of the parameters $R^i_j$ in order for local utility functions to be computed.

5. The ESN must dynamically obtain from agents any additional observable property values it needs to perform (4). This is necessary because some parameters in $R^i_j$ may be dependent on the observable properties of other agents. The set of additional observable properties which must be obtained for this purpose from agents of service type $T^i$ is defined as $O^i_0$.

The above updates (steps 3, 4, 5) can take place either periodically, or whenever an observable property/parameter changes. This is an application specific issue that is decided depending on the amount of communication that would be present in each case.

2.2 Group management

Once an initial group has been established based on LEAF, it is now managed by the GroupLog system. A GroupLog system contains concurrently executing agents able to: (1) communicate through interface predicates, and (2) shared state made available to all members of a group. The data structures created by LEAF are now used to initialise GroupLog, and consist of the following:

- Each agent of the group identified in LEAF is now a member of a GroupLog group and achieved by calls to create_instance($<$agent_instance$>$) predicate. The add predicate is used to support agent instances entering groups.

- The interface of an agent is now encoded using the interface($<$list_interf_pred_names$>$) predicate, where $<$list_interf_pred_names$>$ is a list of interface predicate names and represent service types managed by an agent. At each point in a computation, the agent instance has a current configuration. Given the
current configuration of an agent and an invoked interface predicate there is a set of clauses which are enabled by that configuration and which define the agent behaviour. An instance of an agent only handles one request for communication at a time, but it can launch multiple internal concurrent threads to process the received communication requests.

- The shared state of a group managed by an ESN will contain a collection of tuples. These represent the state of each agent on the completion of LEAF, and the utility information managed by the ESN. The following tuples are used:

  Tup1=global_utility_function(F)
  Tup2=service_types(List_Service_TYPES)
  Tup3=observable_properties(Agent_ID, Properties)
  Tup4=service_attributes(Service_Type, List_Properties)

The number of tuples depend on the parameter types that must be made publicly available by each agent and group.

![Diagram](image)

Figure 1: Sharing of data structures and agent properties between LEAF and GroupLog

Figure 1 illustrates how data structures are transformed between LEAF and GroupLog. The properties of each LEAF agent are now made public via the shared space, and a `begin` predicate may be used to initiate execution within a group. A `GroupLog` group hides its members from the outside but allows the redirection of communication to them through the group interface predicates. The internal concurrency in a group is explicitly defined by its members that have access to a shared group state, for internal coordination. An agent can belong to one or more groups and inspect their membership through predefined predicates. Besides using the interface predicate of other agents, each member
can communicate with other members in the same group through the following
predicate, which provides a form of multicast:

\[
\text{all}(\langle \text{a\_group} \rangle, \langle \text{int\_pred\_name} \rangle, \langle \text{type} \rangle, \langle \text{ans} \rangle)
\]

where \( \langle \text{a\_group} \rangle \) is an instance of a group and \( \langle \text{int\_pred\_name} \rangle \) is an inter-
face predicate. All group members are addressed such that they have defined
\( \langle \text{int\_pred\_name} \rangle \) as an interface predicate. The third argument \( \langle \text{type} \rangle \) is one
of (none, one, quorum, all) and it means the invoker requires, respectively, no
answer, one answer, a quorum, or answers from all the addressed group
members. The fourth argument \( \langle \text{ans} \rangle \) is a list of pairs \( \langle \langle \text{instance} \rangle, \langle \text{reply} \rangle \rangle \rangle \) such
that \( \langle \text{instance} \rangle \) identifies the invoked agent, and \( \langle \text{reply} \rangle \) is the computed so-
lution to \( \langle \text{int\_pred\_name} \rangle \). The use of multicast allows agents to minimise the
number of messages they need to exchange with each other.

3 Example: Resource Sharing in Computational Grids

We use an example to illustrate the two phases of our approach. Consider an ap-
plication utilising resources (computational and data) in multiple administrative
domains. We assume that such an application requires multiple computational
and data resources to execute successfully and there is no one domain that
offers all of these resources. This application may be divided into a number of
tasks (with dependencies defined using a task graph) all of which must be
executed to successfully complete the application. Resource domains may be
developed based on an IP sub-mask, or as logical domains implemented using
middleware such as Globus [6]. In this example, each administrative domain
can offer multiple resources and is represented as a group managed by an
ESN; hence, we can perceive different groups as \( \text{esn}(D_i) \), which represents the
\text{esn} node for the \( i^{th} \) group. We assume that there are ‘\( s \)' \( (s \geq 1) \) participants
in each group. We also consider that each resource owner obtains a monetary
benefit by providing access to their resource by external tasks. The greater
the utilisation of the resource, the greater will be the monetary profit that the
resource owner can collect over a given period of time. Hence, initially resource
providers within a particular group are interested in maximising the utilisation
of their resources, and therefore may act in a selfish manner. However, if an
application manager were to make collective use of resources within a particular
domain, the combined monetary reward for the entire group will improve.

Resource owners may also wish to participate in multiple logical domains,
evaluating the utility assigned by the \text{esn} managing the group. Hence, an
\( \text{esn}(\text{domain}A) \) is composed of different resource providers, and each one of
fers a particular role (service type) in the group (eg data provider, processor
provider, etc). The group as a whole will have a global utility, along with a
local utility for each member of the group. The \text{esn} calculates local utility for
each member of the group based on the observable properties revealed to the
\text{esn} by each provider (this includes parameters such as number of jobs processed
(throughput), time per job (response time), and utilisation). Combining these
three parameters (with a greater emphasis on utilisation) into a utility measure, allows the $esn$ to assign a reward to the resource provider for each application (or collections of applications) that the resource provider participates in. Each resource provider, on the other hand, will also aim to choose application tasks that are likely to improve its local utilisation thereby increasing its reward.

The utility values assigned by each group are then aggregated by the resource provider, to determine its current local utility. Utility value may be provided by the $esn$ for a particular group either for every task executed by the resource provider, or for a collection of tasks. In this case, coordination is achieved by the resource provider deciding which group to join based on rewards received and is therefore implicit based on the decision taken. Co-ordination decisions are made at this stage to form groupings of providers who can work together more effectively (and hence get better rewards from the $esn$). Once these decisions have been made, what results is a “team”, that may now work more effectively together – this is phase 1 of our approach, and achieved through LEAF.

Once a team has been formed, it is now important to enable team members to share information more readily. To achieve this, each team member must register itself and some of its properties into a shared state. Communication between team members is now facilitated through the shared space (via a list of common predicates) or via direct communication (via common predicates maintained by each participant through their interface). Code fragment 1 illustrates how a resource provider group is defined. Figure (2) illustrates the organisation of a domain. In this second phase, group members have a greater trust in each other, and operate in a more cooperative manner.

```prolog
group domain(ID) {  
  context().
  interface().
  domain(ID) : add(compute_single(Idp)) :- domain(ID).
  domain(ID) : add(compute_parallel(Idc)) :- domain(ID).
  domain(ID) : add(data_tape(Idl)) :- domain(ID).
  domain(ID) : add(data_parallel(Idr)) :-
    domain(ID): all(domain(ID),begin(ID),none,_).
}
```

Code Fragment 1: A Game Structure

Discussion

The actions taken by a resource provider are dependent on the environment within which the provider (and the $esn$) are situated. It is likely that the same action taken by a provider, within the same group, may lead to different rewards. Hence, each provider within a team may base its involvement on an expected reward after a given number of task executions. Our analysis above assumes that the provider commits to participate in a number of concurrent applications (and would base its reward on this assumption). It is also possible, however, for a provider to leave midway through the execution of an application, and
therefore the expected reward must be calculated over a finite horizon. There are two particular scenarios associated with this: (1) a provider leaving does not have an impact on the other participants in the team, (2) a provider leaving causes an impact on the other providers in the team and hence the ESN has to adjust reward for other participants. Case (2) can therefore lead to unbalanced teams and the ESN must decide how rewards must be given to agents within this context.

The add and delete predicates in GroupLog can be used to support providers leaving and entering groups. Once a new provider joins a group, others are made aware of its presence via the shared space.

4 Conclusion and Further Work

A mechanism to support co-ordination in dynamic multi-agent communities is provided, based on a two phased approach. The first phase involves co-ordinating selfish agents, which try to maximise their local utilities. The agents reveal very little about their internal state at this stage, and have imprecise knowledge of other participants. The second phase involves co-operative agents, which are able to reveal more details about their internal state, and are able to use a shared space to exchange data. The first phase is implemented using the LEAF system, and the second using the GroupLog system. Interaction between these is achieved using a set of common data structures which enable the structure being created through LEAF to be instantiated as a GroupLog group. At present, these two systems exist on their own, and a stronger coupling and interaction between them is being implemented. GroupLog has been implemented using PVM-Prolog [4], and existing work includes creating a Java interface to this. LEAF has been implemented using Java, and builds on the FIPA-OS implementation from Emorphia. LEAF is also being deployed within the AgentCities consortium [1].
Another aspect being explored is the time period over which a group must remain a factor dictated by the type of environment within which the group exists. Our scenario's range from having the group active over a finite horizon (limited number of states), to an infinite horizon (the group structure does not change over the total lifetime of all agents in the group). The exact factor that determines which of these two schemes is most appropriate is not explored here. A simple example based on resource sharing in a Computational Grid is used to demonstrate the concepts involved although the ideas presented here are applicable in a number of other domains.

References