A Dynamic System for Distributed Reasoning

Jiefei Ma and **Krysia Broda** and **Alessandra Russo** and **Keith Clark**
Department of Computing, Imperial College London
{jm103, kb, ar3, klc}@doc.ic.ac.uk

Abstract

This paper proposes a multi-agent, multi-threaded architecture for a *distributed inference* system for a *dynamic* group of agents. The system can opportunistically make use of new agents that join the group, whilst a proof is in progress. It can also recover if an agent leaves. Final proofs only make use of the knowledge bases of agents that are in the group when the inference is concluded, and are sound with respect to their combined knowledge. The group can be simultaneously engaged in multiple independent proofs.

We show how the same architecture can support both distributed definite clause logic programming, and distributed abductive reasoning using negation-as-failure.

Both the architecture and the two proposed inference system applications have been implemented using a multi-threaded distributed Prolog system, Qu-Prolog.

Keywords: multi-agent system, multi-threaded distributed inference, abduction, Qu-Prolog.

Introduction and Motivation

The focus of this paper is a multi-agent multi-threaded architecture for *open distributed* inference systems where knowledge and constraints are distributed over a group of agents that cooperate to produce a proof. The architecture makes use of the multiple threads and high-level communication features of Qu-Prolog to support a cooperative working environment for both distributed deductive (Robinson, Hinchley, & Clark 2003) and abductive reasoning (Ma *et al.* 2007). Each agent has its own knowledge base and consistency constraints, and is assumed to be cooperative and trustworthy. Different agents may contribute to the computation of a collective proof, but each of their sub-proofs *must* satisfy the relevant consistency constraints of *all* the agents who *have contributed to the proof*. We call this subset of the agents in the group, who have contributed, the proof *cluster*. To allow for an *open* system where agents can join and leave the group at will, the architecture is shown to support a distributed abductive inference algorithm, that can opportunistically make use of new agents that arrive whilst a proof is in progress, dynamically extending the current proof cluster. It can also recover if a contributing agent leaves the group, dynamically reducing the current cluster and discarding any sub-proofs to which the agent may have contributed.

The architecture is implemented using the multi-threaded distributed Qu-Prolog (Clark, Robinson, & Zappacosta-Amboldi 1998). Each agent in the open group of all the agents that may participate in a proof is implemented as a separate Qu-Prolog process that can be distributed over a network of hosts. Each Qu-Prolog process comprises multiple-threads allowing it to be participating in multiple (abductive) proofs at the same time. Communication between threads in the different agent processes uses a network demon, Pedro (Robinson 2007).

To illustrate the flexibility and generality of the architecture, two example applications are then described: distributed definite clause inference and distributed abductive inference making use of negation-as-failure. We briefly discuss a use of the second system for distributed abductive planning. Although quite simple, the distributed abductive reasoning example shows the dynamic nature of the architecture. At any moment in time the current cluster of agents collaborating in trying to find the abductive proof can change. This is not only as a result of their respective constraints but also because agents can leave and join the wider group of available agents, and hence the current proof cluster, whilst the proof is in progress.

The next section presents the architecture in detail. We then describe two types of collaborative inference system supported by the architecture, definite clause logic programming and abduction (Kakas, Kowalski, & Toni 1992) with negation-as-failure (Clark 1978). We conclude with a brief discussion of related work and ideas for future work.

Architecture

This section describes the architecture and main features of the system. This is an *open distributed inference* system that supports collaborative proof of a query given to or internally generated by any agent in the system. Each agent is equipped with its own *knowledge base* (definite or normal logic program) and a (possibly empty) set of integrity constraints. The collection of agents in the system can dynamically change during a particular inference process. This can result in the agents involved in a proof, the current *proof cluster*, to change. The agents can have overlapping or disjoint knowledge bases. Two knowledge bases are disjoint if

they do not share the same clause. The system assumes communication between agents to be safe and reliable, namely the messages sent between two agents cannot be lost or corrupted, and each agent is rational and trusted by the others. As its main purpose is to support coordinating collaborative reasoning, the handling of various network attacks or fatal network failures is currently not considered. Currently, the system does not allow for the possibility of malicious agents interfering in the collaboration between other agents.

Architecture Overview

A query can be submitted to any of the agents, A_i, in the group, which will return the answer. The reasoning process starts within agent A_i. It tries to construct an answer using only its own knowledge. But during its reasoning process, it can "ask for help" from other agents in the group. The query answer returned by the agent is associated with the *cluster* of all the agents that have *contributed* to its proof.

The main features of the architecture are its high level inter-agent communication and the internal concurrency of the agents. The agents comprise several distinct time-shared threads of computation that co-ordinate via internal thread-to-thread messages and a shared blackboard. The agents can be distributed over a network of host computers allowing the different agents that can potentially help with a sub-proof to be concurrently searching for a proof.

Inter-Agent Communication

Inter-agent communication is via asynchronous message-passing, using KQML (Finin *et al.* 1994) performatives. Messages are communicated between the agents using a communications demon, Pedro (Robinson 2007), which routes messages with a specified agent destination to that agent. An agent identity has the form `agentName@host`. A thread named `Th` within an agent has an identity of the form `Th:agentName@host`. Messages can be addressed either to the agent or to a specific thread within the agent. If the former, the message is sent to the initial agent thread, else it is routed directly to the named thread within the agent. All threads have their own message buffer of received unread messages. Pedro will also forward messages posted to it without a specified agent destination using lodged message pattern subscriptions. We use this to give some of the functionality of a KQML matchmaker(Kuokka & Harada 1995).

When an agent is launched it first connects to the Pedro demon. Its name `agentName` and host name `host` are recorded by Pedro so that Pedro can route to the agent all messages addressed to it or one of its threads. The agent then posts an application specific `register` message to Pedro containing its identity to inform the agents already in the group of its arrival. This message will be forwarded to all existing agents because they will have subscribed for such `register` messages. The new agent then lodges a subscription with Pedro for both `register` and `unregister` messages, so that it will become aware of new agents that join afterwards, and when an agent leaves. Finally, it posts `advertise` messages to Pedro announcing the predicates of conditions for which it is willing to try to find proofs, and it subscribes for such advertisements posted by other agents that mention predicates for which it may need proof help.

There is one key difference between Pedro and a standard KQML matchmaker. The latter will remember advertisements as well as subscriptions for advertisements, whereas Pedro only remembers subscriptions. This means that whenever an agent receives a `register` message, indicating a new agent has joined the group, it must send its advertisements directly to the new agent. Each agent maintains its own "yellow pages" directory of other agents currently in the group, and the directory only records the advertisements. We used Pedro and local directories instead of a signle KQML matchmaker in order to reduce network traffic when an agent tries to find helpers.

Internal Architecture of Agents

The agents are linked in an acquaintance network by means of the local "yellow pages" directories within each agent. Using its directory an agent can find suitable helper agents and request them to provide it with subproofs, as the need arises. When asked, the helper agent will return the computed answers (if any) to the requesting agent, one at a time.

Each agent can be involved in several proofs at the same time. They are *multi-threaded* and *multi-tasking*. Specifically, each agent has a *Coordinator Thread* (CT), for handling proof requests, a *Directory Thread* (DT), for maintaining the "yellow pages" directory, a number of *Worker Threads* (WT) managing different proof tasks concurrently and independently, each of which is linked to a *Reasoning thread* (RT). In addition, there can be several *Broker threads* (BT) spawned by the RTs to proofs.

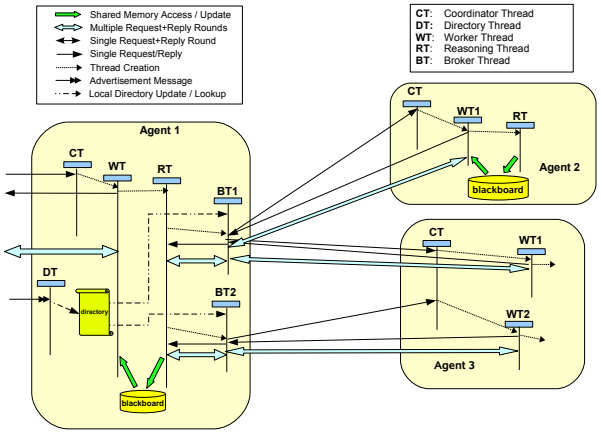

Figure 1: Agent Internal Architecture

In summary, the architecture of each agent supports three main functionalities: maintaining the directory, handling incoming proof requests and requesting help from other agents. These are described in detail in what follows.

Maintaining the Local Directory The DT is a persistent thread responsible for maintaining the agent's "yellow

pages" directory. It performs the following tasks:

1. It starts by performing the initialisation interaction with the Pedro server described above: connection, and the posting of the agent's `register`, `subscription` and `advertisement` messages. Messages forwarded by Pedro to the agent will be sent to the DT thread.

2. Whenever DT receives a `register` message from another agent, it sends its advertisements to that agent. If it receives an `unregister` message from another agent, it removes from its directory all that agent's advertisements.

3. It updates its local directory in response to `advertise` or `unadvertise` messages from another agent, whenever they are received.

Handling Incoming Requests As mentioned above, the incoming proof requests are handled by the co-ordinator thread, CT. To allow multiple reasoning tasks to run simultaneously, for each *accepted* incoming request CT spawns a worker thread WT. An incoming request includes the specific goal to be proven, the identities of all agents in the current cluster, and the current proof information. If the agent is in the current cluster, the request is always accepted. If not, some of the proof information may need to be checked by the agent before the agent accepts the request.

A spawned WT immediately spawns a reasoning thread RT to perform the reasoning task. It then sends a `ready` message to client agent. More specifically, it sends the message to the thread within the client agent that sent the proof request. As explained below, this will be a broker thread within the client agent, and all communication regarding the proof task is between the WT thread and this broker thread.

An RT may seek help from any number of other agents via *Broker Threads* (BT) that it creates. It also eagerly generates all the answers for the sub-goal it is proving and stores them in generation order on a *blackboard* internal to the agent (see Figure 1). The WT waits for and services `next` requests from its client agent thread. On receipt, it removes the next result from its RT from the blackboard, if available, and sends it to the client thread, or waits if the next answer has not yet been found by its RT. The RT informs its WT when it has explored all proof paths, and no more answers will be found. At this point the WT will respond with an `eos` message to its client when it receives a `next` request.

The rationale for having a producer-consumer relationship between the WT and its RT using a blackboard as a buffer is to isolate the reasoning process from the client agent. The RT just searches for *all* possible answers for its given reasoning task, adding each to the blackboard as it is found, independently of its use by the client. There is one exception. If the WT receives a `discard` message from the client thread, it erases any unused answers placed on the blackboard by its RT and terminates the RT if it is still running. It is worth noting that after a reasoning thread is asked to terminate, it will send a `discard` message to all its active broker threads (see next section), which will in turn forward this to the WTs in the server agents that are finding sub-goal answers for them.

Getting Help From Other Agents This section describes in detail the interactions when an agent asks for sub-proof help. Whenever an RT asks for help, it creates a BT to handle all the request-response communications with helper agents. BT is given the goal to outsource, the identities of the agents in the current cluster, and any other necessary proof information. Each RT maintains the cluster set of agent identities for the current state of its proof.

BT checks the advertisements in the agent's "yellow pages" directory to create a helper list. It then sends a request to the CTs of all the agents in the helper list giving the proof information including goal and identities of all agents in the current cluster. An implicit proof contract with a helper agent is established as soon as the BT receives from that agent a `ready` message, sent by the WT within the agent that will have been created for the proof task. BT then sends that WT a `next` message to ask for its first answer. Usually it will be able to send `next` messages to several helper agents. It then waits for the first proof to be returned by any of these helpers. When the helpers are on different hosts, this is an or-parallel search for alternative proofs.

Whenever the RT wants an alternative result for an outsourced sub-goal `G`, (either for the first time or during backtracking), it sends a `next` message to the BT handling answers for `G`. The BT searches its message buffer for an answer from any helper agent. These answers will have been inserted into its message buffer as and when they arrive, so the answers from each of the different helper agents will be interspersed in the buffer. There are three cases:

- A `tell` message is found and removed containing an answer. BT extracts the answer from the message and forwards it to the RT. BT then sends a `next` message to the helper agent WT thread which sent the message, so that if there is another answer from that agent, it can be returned and buffered in BT's message queue.
- An `eos` message is found. BT removes the sender from its list of helpers. When this list becomes empty, BT sends an `eos` message to its RT.
- There is no message in the buffer, but the helper list is non-empty. BT suspends until a message arrives.

Each `tell` message will contain the possibly instantiated sub-goal, a possibly augmented proof information, and a list of agent identities that should be added to the cluster set if this answer is used. This list will be non-empty if the helper agent is a new agent not in the cluster set passed to it in the sub-proof request, and if it has itself requested sub-proofs from agents not in the given cluster set.

Finally, each time the BT services a `next` request from its RT it checks to see if the local "yellow pages" directory contains the identity of any agent not on the current helper list that has since advertised ability to help with the sub-goal BT is handling. If so, this agent is added to the helper list and sent a sub-goal proof request allowing this late entrant to contribute candidate proofs.

The rationale for having a separate BT to handle the communications is to isolate the RT from the helper agents. BT forwards answers to its RT in the order that they arrive from the different helper agents and merges the answers. From

the RT's point of view, a BT is an internal thread that can provide it with alternative proofs for a sub-goal as they are needed. RT can have several different BTs active and buffering results for different sub-goals. Whenever the RT backtracks to a sub-goal with an associated BT, it asks the BT for the next available result. If no more results are available (i.e. the `eos` message is returned by the BT), RT fails that sub-goal and continues the backtracking.

Each BT exits automatically after sending the `eos` to the RT. It is worth noting that a BT can have in its buffer a maximum of one answer from each helper agent since it sends the `next` request to a helper agent only after it as forwarded that helper's previous answer to its RT. Of course, independently the RT threads within each helper agent are finding all the answers and caching them on their internal blackboards. In this way the system keeps network traffic to a minimum whilst still maintaining the reasoning performance since the RT can continue its reasoning when the BT is fetching the next result. Finally, whenever an RT of an agent is terminated by its linked WT, RT itself sends a `discard` message to all the BTs it has created. Upon receiving such a message, each of these BTs will forward the `discard` message to all the current helper agents.

Applications

Various distributed algorithms can be implemented as meta-interpreters run by the agents, for example, deductive inference among different agents' knowledge, hypotheses generation by a set of agents and planning between robots.

Distributed Deductive Inference with Definite Logic Programs

Distributed deductive inference without negation-as-failure (Clark 1978) can be implemented easily with the system. In this case, we assume that an agent's knowledge base is a definite logic program, and an agent makes an advertisement of each predicate that has a definition (possibly empty) in its knowledge base. During an inference process, when an agent is unable to resolve a sub-goal (e.g. it doesn't have a definition for it in its knowledge base), it can look up from the directory a list of potential helper agents that know the sub-goal, and outsource the sub-goal to these agents. Once an answer is returned, the agent uses it straight away and continues to resolve the rest of the sub-goals.

It is possible that two agents advertise the same predicate. To avoid cyclic outsourcing of the same sub-goal, the helper agent is also given a list (called the *Don't ask* list) of agent identifiers. The list typically contains the agent who outsources the goal, and other agents who have been outsourced by that agent for the same goal. Therefore, the helper agent should not "ask" anyone in the list to resolve the given goal.

Negation as Failure and Abductive Reasoning

It has been shown that standard negation-as-failure (NAF) can be simulated by abductive inference (abduction) (Eshghi & Kowalski 1989). This is also true for distributed deductive inference with NAF, where each agent's knowledge is a normal logic program and the reasoning can be simulated by distributed abductive inference. A distributed abductive algorithm, which allows agents in the system to perform collaborative abductive inference, was proposed and implemented in (Ma *et al.* 2007). It is assumed that all the agents agree on the language and the set of abducibles. An agent can also have integrity contraints in its knowledge as well. Agents can contribute to a proof and form a *cluster* dynamically. The algorithm guarantees that the final result obtained is consistent with the set of integrity constraints of the agents in the associated cluster.

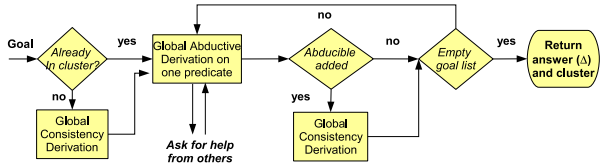

Figure 2: Global Abductive Derivation

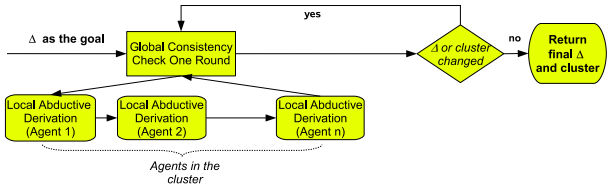

Figure 3: Global Consistency Derivation

The algorithm is based on Kakas-Mancarella abductive proof procedure (Kakas & Mancarella 1990), and consists of four derivations – *Global Abductive Derivation (GAD)*, *Global Consistency Derivation (GCD)*, *Local Abductive Derivation (LAD)* and *Local Consistency Derivation (GCD)* (see Figures 2 and 3). During the GAD or GCD, an agent is allowed to ask for help with resolving or failing a sub-goal. The answer returned from a helper agent also includes the set of hypotheses (Δ) that are consistent with every agent in the cluster. So, when an agent outside the current cluster is outsourced a sub-goal, it has first to check whether the Δ passed with the sub-goal is consistent with its knowledge. If it is, then the agent joins the cluster and continues with the proof; otherwise it is not added to the cluster. Whenever a new abducible is added to Δ, it must be passed around the agents in the current cluster to check for consistency (i.e. starts a GCD). The *Don't ask* technique for avoiding cyclic outsourcing of the same sub-goal may still be used.

Consider an example of distributed abduction with 3 agents. The set of abducibles is $\{c, d, e\} \cup$ *negative_literals*, and the agents' knowledge and directories are:

	agent 1 (A1)	**agent 2 (A2)**	**agent 3 (A3)**
knowledge	a :- b1, b2. ← c, **not** e.	b1 :- c. b2 :- **not** e.	b1 :- d.
directory	advertised(agent2, b1). advertised(agent2, b2). advertised(agent3, b1).	advertised(agent1, a). advertised(agent3, b1).	advertised(agent1, a). advertised(agent2, b1). advertised(agent2, b2).

When A1 is asked to explain a, the following steps happen:

1. initially, only A1 is in the cluster, and the Δ is empty.

2. In the GAD, A1 resolves a to get $b1, b2$. It then outsources the $b1$ to A2 and A3. A2 and A3 then independently and concurrently try to explain $b1$.
3. **(a)** A2 joins the cluster and explains $b1$ by abducing c. The GCD starts and $\Delta = \{c\}$ is passed to A1 (because it is in the cluster) for a consistency check[1]. In order to make c to satisfy its integrity constraints, A1 first abduces c and then abduces e. $\Delta = \{c, e\}$ is then passed to A2. A2 agrees on it without expanding it so the GCD terminates. An answer of $b1$ with $\Delta = \{c, e\}$ and the cluster $\{A1, A2\}$ is returned to A1.
(b) A1 continues its proof by attempting to show $b2$ in the GAD. However, it can't resolve $b2$ so it outsources $b2$ to A2. A2 can't explain it either because **not** e is inconsistent with the given $\Delta = \{c, e\}$. So A2 fails the proof and then A1 backtracks and waits for another answer for $b1$.
4. **(a)** Alternative to 3(a), A3 joins the cluster (which has A1 only) and explains $b1$ by abducing d. Again, the GCD starts and A1 needs to check $\Delta = \{d\}$. A1 agrees with it without adding new abducibles. So the proof result with $\Delta = \{d\}$ and the cluster $\{A1, A3\}$ is returned.
(b) as in 3(b), A1 continues its proof with $b2$ by outsourcing it to A2. A2 first checks whether it agrees with $\Delta = \{d\}$. Fortunately, it does without expanding Δ, and can also explain $b2$ by abducing **not** e. Again, the new $\Delta = \{d, \mathbf{not}\ e\}$ is passed to A1 and A3 to check during the GCD. Clearly, they all agree. Hence the new Δ and new cluster (containing all the agents) are returned to A1.
5. finally, A1 obtains a proof for the goal a consistent with the knowledge of all the agents.

Thanks to the multi-tasking support from the system, 3(a) and 4(a) are actually performed in parallel. This is because each agent is multi-threaded and *agent1* could be involved in the two GCDs and the GAD (i.e. waiting for the outsourced proofs in this case) at the same time.

In addition, the system is open such that agents may join and leave the system during a proof. For example, if an agent A4 with knowledge `b2 :-` **not** `f` were to join the system before step 3(b), then the execution of GAD in 3(b) would succeed because on arrival of A4, `advertised(agent4, b2)` would be added to each agent's directory. A1 could have then outsourced $b2$ to A4 in 3(b), which would have given a final $\Delta = \{c, e, \mathbf{not} f\}$.

Distributed Abductive Planning

We just saw how agents in the system can perform abductive reasoning collaboratively. In fact, a more interesting application is to have agents do distributed planning via abduction. For example, in a cooperative robot environment, each robot may have its own knowledge about the environment and may perform its own set of actions. In order to generate a plan to achieve a goal state, each robot can advertise its ability. The planning process invokes a consistency check of the intermediate plans among the robots to guarantee the final plan obtained is valid (i.e. it doesn't lead to states where a robot's integrity constraints are violated). A simple distributed abductive planner (Ma 2007) inspired by the simple abductive event calculus planner (Shanahan 2000), has been implemented and tested with the system. The execution of the planner is very similar to that of the generic distributed abductive algorithm. It separates the hypotheses Δ into two sets – the set containing the positive abducibles of `happens(Action, T)` and `before(T1, T2)`, and the set of negative goals (e.g. **not**`(clipped(Fluent, T1, T2))`). Whenever a `happen` atom is abduced, it invokes the NAF check on negative goals amongst the agents.

There are several advantages of this planning system: (1) only the robots with useful knowledge or skills will be recruited for the plan; (2) several robots may compete in finding and offering sub-plans for a given intermediate goal at the same time; (3) robots with new information or skills may join the system easily to "rescue" a plan.

Related work

The system is implemented in Qu-Prolog (Clark 2007), which has the occurs check in its unification algorithm and was developed explicitly for implementing sound first order inference systems. Qu-Prolog has been used to implement agent based co-operative inference systems for full first order predicate logic (Robinson, Hinchley, & Clark 2003) and (Zappacosta 2003).

Other related cooperative agent-based systems include (Hunter, Robinson, & Strouper 2005) for program verification which makes use of broker agents to find agents with appropriate expertise for specialised sub-proofs, and (Franke *et al.* 1999), which links hybrid theorem provers together using KQML messaging.

As for the distributed abductive reasoning algorithm, the only related system is ALIAS (Ciampolini *et al.* 1999). As far as we understand the ALIAS system, and its extension, which is coupled with the LAILA language (Ciampolini *et al.* 2001) for co-ordinating abductive reasoning amongst a group of agents, and expressing the knowledge of each agent, these are some key differences:

- In ALIAS, each bunch of agents use a shared space called blackboard to facilitate communications of hypotheses among them. In our system, the hypotheses are passed between agents via peer-to-peer communications, i.e. no centralised space and the agents are highly decoupled.
- The acquaintance relation between ALIAS agents is specified in each agents' *Agent Behavior Module* by explicit annotations of sub-goals in the LAILA rules which specify which other agent or agents should be queried for solutions. In our system, the agents that can be asked to prove a sub-goal are determined when that sub-goal needs to be solved using a local directory of agents who have advertised their willingness and capability to help.
- When an ALIAS agent abduces a new hypothesis, it has to ask all the agents in the bunch of co-operating agents (in our terms the cluster) to check for consistency. But during the consistency check, an ALIAS agent cannot "ask" another agent for help to keep the current hypotheses consistent by expanding it. In our system, it is possible.

[1] This is equivalent to asking A1 to explain c. See (Ma *et al.* 2007).

- We believe that an ALIAS bunch can only be actively pursuing one distributed inference at a time, whereas in our system there can be several proofs being pursued concurrently inside each agent.
- ALIAS does have the concept of rambling agent that can join an existing bunch. However, the new agent does not join to help with a sub-goal of a current proof. Also, when the rambling agent exits a bunch, key knowledge that it has contributed is not retracted.

Discussion and Future work

This paper describes an open multi-agent, multi-threaded architecture that can support distributed deductive and abductive reasoning. The system is *open* in that it allows new agents to join or leave the group as they wish at any time.

Within the distributed abductive inference, the answers for a collective proof can come from different agents but they are guaranteed to be consistent with the integrity constraints of all the agents who have contributed to the proof. Each agent is multi-threaded, allowing the system to handle queries and perform reasoning tasks concurrently.

The system was tested on several Linux PC with $3.0GHz$ processor and $1GB$ RAM. In order to test the system in a harsh environment, we have also ported Qu-Prolog 8 to the *Gumstix computers*[2], which have the size of a chewing gum. They have a $400MHz$ processor, $64MB$ RAM and $16MB$ ROM and run ARM Linux. The Gumstix computers can connect to a 802.11 Wireless LAN with suitable expansion boards. This allows us to explore applications of the system whereby some agents are on Gumstix computers, with perhaps quite simple knowledge bases. Others with larger knowledge bases and the Pedro server can be hosted on PCs on the same Wireless LAN. This type of configuration is particularly suitable for applications such as multi-sensor or multi-robot co-operative sense data interpretation, and multi-robot planning.

The current meta-interpreter implementation of the algorithm presupposes that an agent in a given cluster does abductive reasoning whenever it has the appropriate reasoning capability. Other agents are asked for help only when the local derivation has failed. A possible extension/variation of this meta-interpreter is to allow for *lazy* agents. These are agents that even though they have appropriate reasoning capability for answering a given query, opt to ask for help to other agents instead of undertaking their local derivation. Of course appropriate heuristics will be needed as to when an agent should/could be lazy and when not, to guarantee progress of the computation as well as not network overloading with large number of messages between agents.

The system has several potential applications such as the illustrated multi-agent reasoning and the previously mentioned multi-robot planning and collaborative interpretation of sense data. Future extension of this work includes the development of specialised meta-interpreters tailored to the particular domains of application.

[2]http://www.gumstix.org

Acknowledgements

The first author is supported by research continuing through participation in the International Technology Alliance sponsored by the the U.S. Army Research Laboratory and the U.K. Ministry of Defence.

References

Ciampolini, A.; Lamma, E.; Mello, P.; and Torroni, P. 1999. Rambling abductive agents in alias. In *Proc. ICLP Workshop on Multi-Agent Sytems in Logic Programming (MAS'99)*.

Ciampolini, A.; Lamma, E.; Mello, P.; and Torroni, P. 2001. LAILA: a language for coordinating abductive reasoning among logic agents. *Computer Language* 27(4):137–161.

Clark, K. L.; Robinson, P. J.; and Zappacosta-Amboldi, S. 1998. Multi-threaded communicating agents in Qu-Prolog. In Toni, F., and Torroni, P., eds., *Computational Logic in Multi-agent systems*. LNAI 3900, Springer.

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Databases*. Plenum press. 293–322.

Clark, K. 2007. Multi-agent systems lecture notes. Introduction to QuProlog 7.4.

Eshghi, K., and Kowalski, R. A. 1989. Abduction compared with negation by failure. In *ICLP*, 234–254.

Finin, T.; Fritzson, R.; McKay, D.; and McEntire, R. 1994. KQML as an agent communication language. In *Proceedings 3rd Int. Conference on Information and Knowledge Management*.

Franke, A.; Hess, S.; Jung, C.; Kohlhase, M.; and Sorge, V. 1999. Agent-oriented integration of distributed mathematical services. *Universal Computer Science* 5(3):156–187.

Hunter, C.; Robinson, P.; and Strouper, P. 2005. Agent-based distributed software verification. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science*, volume 24 of *ACM International Conference Proceeding Series*, 159–164.

Kakas, A. C., and Mancarella, P. 1990. Database updates through abduction. In McLeod, D.; Sacks-Davis, R.; and Schek, H.-J., eds., *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, 650–661. Morgan Kaufmann.

Kakas, A. C.; Kowalski, R. A.; and Toni, F. 1992. Abductive logic programming. *Journal of Logic and Computation* 2(6):719–770.

Kuokka, D., and Harada, L. 1995. On using KQML for Matchmaking. In *1st International Joint Conf. on Multi-agent Systems*, 239–245. MIT Press.

Ma, J.; Russo, A.; Broda, K.; and Clark, K. L. 2007. Dare: A system for distributed abductive reasoning. *JAAMAS (accepted for publication)*.

Ma, J. 2007. Distributed abductive reasoning system and abduction in the small. Technical report, Department of Computing, Imperial College London.

Robinson, P. J.; Hinchley, M.; and Clark, K. L. 2003. QProlog: An Implementation Language with Advanced Reasoning Capabilities. In Hinchley et al, M., ed., *Formal Appraches to Agent Based systems, LNAI 2699*. Springer.

Robinson, P. J. 2007. Pedro Reference Manual. Technical report, http://www.itee.uq.edu.au/˜pjr.

Shanahan, M. 2000. An abductive event calculus planner. *Journal of Logic Programming* 44(1-3):207–240.

Zappacosta, S. 2003. Distributed implementation of a connection graph based on cylindric set algebra operators. In Hinchley et al, M., ed., *Formal Appraches to Agent Based systems, LNAI 2699*.