Software Agent Technologies

Authors: Hyacinth Nwana & Michael Wooldridge (Manchester Metropolitan University)

Approved By:

-----------------------------------------------------------
Robin Smith
Nader Azarmi

Authorised By:

-----------------------------------------------------------
Nader Azarmi (Project Manager)

MIDAS

Intelligent Systems Research
Advanced Applications and Technology

Abstract

It is by now a cliché that there is no one, universally accepted definition of intelligent agent technology, but a number loosely related techniques. And yet there are certain themes that appear common to agent-based systems, and correspondingly, certain problems that must be addressed and overcome by all agent system builders. The aim of this paper is to briefly survey the tools and techniques that can be used to address these common issues, and that hence form a substrate for software agent systems. We begin with a review of agent communication languages, focusing particularly on the emerging standard known as KQML. We then present a thumbnail sketch of various programming languages for building agent-based systems, and go on to discuss support for ontologies, which allow agents to communicate using commonly-defined terms and concepts. We then consider other computing infrastructure support for agent-based systems, in particular, the use of client-server architectures and distributed object frameworks. Finally, we present some general comments and conclusions.

1 Introduction

At the time of writing, intelligent agents and multi-agent systems are among the most rapidly growing areas of research and development in computer science. Unfortunately, as with object technology a decade ago, there is great confusion about agents: hardly anyone agrees on even the basic question of what an agent is, still less on more contentious issues. New definitions and systems seem to appear one week, and disappear the next. Journalists and practitioners, ever eager to keep abreast of new technologies, attend conferences and seminars only to be presented with a collection of apparently unrelated concepts and terms. And of course, experience with all things object-oriented indicates that this situation is here to stay, at least for the foreseeable future.

Given this state of affairs, it would be naive to make predictions about how agent technology will develop, and in particular, about the exact form or structure of future agent applications. And yet, there are certain themes that appear common to most agent-based systems, and correspondingly, certain problems that must be addressed and overcome by all agent system builders. In brief, the purpose of this paper is to survey the various technologies that are capable of providing (or which currently provide) this substrate for building agent-based applications. Note that, for the reasons we mentioned above, we do not attempt to survey actual agent architectures or applications. Rather, the aim is to focus on what might be called the “enabling technologies” for agent-based systems.

Figure 1 - Some Technologies for Developing Software Agent Applications
In order to facilitate such an overview, an abstraction of some current or future technologies involved is helpful. Figure 1 shows one classification of some of the technologies involved. We emphasise that it represents an abstract organisation of the technologies that facilitate the implementation of agent-based applications (cf. Mayfield et al., 1996). The hierarchy depicted may be considered by some to be contentious and arbitrary. Nevertheless, it represents a useful classification device, around which the remainder of the paper will be structured.

1.1 The Structure of this Article

Typically, applications containing multiple agents make use of an agent communication language (ACL); the idea is similar to a human society using a common language (such as English). A couple of categories of agent communication languages are reviewed in Section 2. In Section 3, languages that facilitate the construction of agent applications are briefly listed. However, we note that agent-based applications can be (and have been) developed using traditional third generation languages like Lisp, C, or Prolog, and object-oriented (OO) languages such as C++ or Smalltalk. Next, agents working together need to share a certain amount of foundational, “common” knowledge, in just the same way that humans do. When one agent tells another agent to buy a widget, they must agree both on what a “widget” is, and what “buy” means. Section 4 reviews the techniques available for representing such foundational knowledge. In Section 5, we briefly discuss some current and emerging computing technologies that lend themselves to supporting agent-based applications, and section 6 provides a discussion and conclusion to the paper.

2 Agent Communication Languages

A key rationale for having multiple agent systems (MAS) is that the ensemble of agents provide “added value”, beyond that which would otherwise be obtained from any single agent (see Nwana, 1996; Nwana & Ndumu, 1996). This added value is typically achieved via co-operation. If every agent had perfect information about the state of the entire system - if every agent knew what every other agent knew, what every other agent intended to do, and how everything in the system stood in relation to everything else in the system - then co-operation could proceed without any communication at all (see, e.g., Nwana et al., 1996). The agents could work together as a perfectly co-ordinated team, without ever having to communicate. Of course, except in the most trivial systems, such perfect knowledge is impossible to achieve. Agents have at best partial, possibly incorrect information about the state of their environment. Thus, in order to cooperate effectively in any moderately realistic system, agents are required to communicate with one-another. Where agents need to communicate, they must individually “understand” some agent communication language (ACL). Indeed, Genesereth & Ketchpel (1994) maintain that some piece of software

“is a software agent if, and only if, it communicates correctly in an agent communication language” p. 50.

Though we believe this statement is arguable because it fails to mention other attributes that we consider germane to agenthood, we acknowledge the importance of having an ACL: imagine a society without some lingua franca.

ACLs are designed specifically to facilitate communication between two or more agents. Agents need to communicate information, intentions, goals, and so on to other agents. As Cohen & Levesque (1995) also note, an ACL should allow agents to enlist the support of others to achieve goals, to monitor their execution, to report progress, success, failure, to
acknowledge receipt of messages, to refuse task allocations, and to commit to performing tasks for other agents.

Most ACLs, both “standard” and ad hoc, derive their inspiration from speech act theory (Austin, 1962; Searle, 1969), which was developed by linguists in an attempt to understand how humans use language in everyday situations, to achieve everyday ends. In speech act theory, human utterances are viewed as actions, in the same sense as actions performed in the everyday physical world (e.g., picking up a block from a table). The theory considers three aspects of utterances. Location refers to the act of utterance itself; simply uttering an acceptable sentence of some language. Illocution refers to the “type” of utterance. For example, and utterance may be a request to turn on the heater or an assertion about the temperature. There are many different illocutionary verbs (or performatives) in English: examples include “request”, “warn”, “inform”, and so on. Perlocution refers to the effect of an utterance: how it influences the recipient. In the context of agent-based applications, researchers have proposed ACLs based on speech act theory, or more specifically, on illocutionary speech acts.

The illocutionary verbs (such as request, inform, warn, and so on) in a natural language like English typically correspond to performatives in some ACL. There is an implicit assumption that the illocutionary performative utterance of some agent succeeds because the sending agent is communicating some attitude(s) of itself such as beliefs, goals, or assertions asserting. It is illocutionary because the agent expects that its utterance will be understood (as it intends it to be) by the receiving agent, even though it has no control over how the utterance influences this receiving agent - i.e., it is not perlocutionary. The aspect of locution, in the context of ACLs, is trivial because it is usually implicit in the explicitly encoded performative. In other words the type and structure of the message, e.g.,

```
accept [to: agent_2, from: agent_3, reference: contract_4, cost: £40]
```

is implicitly an instance of a locutionary act. More importantly, it is assumed that by virtue of being sent, it will effect some illocutionary action (in this example, agent_2 updates its belief set to reflect the fact that agent_3 has accepted to do contract_4 at the cost of £40).

We shall classify ACLs into two categories: “standard” ones and ad hoc ones.

### 2.1 A Standard ACL - KQML

The Knowledge Query and Manipulation Language (KQML) is an evolving standard ACL, being developed as part of the DARPA Knowledge Sharing Effort (KSE) (Neches, 1994). The KSE is a distributed research program that is investigating and constructing software tools for co-ordination and knowledge sharing among information systems, i.e., between separate knowledge-based modules, as well as between knowledge-based systems and databases (Labrou & Finin, 1994). As Finin (1995) notes, the KSE research program

“focuses on the ability of such agents to effectively inter-operate by communicating information and knowledge in spite of problems introduced by heterogeneity of platforms, implementation technology, operating environments, and development. This communication requires a common language or language framework which involves syntactic, semantic and pragmatic components”.

The KSE programme consists of four separate but inter-working groups. One major product so far from one of these groups, the External Interfaces Working Group, is KQML: a high-
level communication language and protocol for exchanging information and knowledge. KQML has been implemented in several research groups across North America.

At the heart of KQML are more than three dozen performatives that define the allowed “speech acts” that agents may use, and which provide the substrate for constructing more complex co-ordination and negotiation strategies (as discussed in Nwana et al., 1996). These performatives are grouped into nine categories, as shown in Table 1.

<table>
<thead>
<tr>
<th>Category</th>
<th>Reserved performative names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic informational performatives</td>
<td>tell, deny, untell, cancel,</td>
</tr>
<tr>
<td>Basic query performatives</td>
<td>evaluate, reply, ask-if, ask-about, ask-one, ask-all, sorry</td>
</tr>
<tr>
<td>Multi-response query performatives</td>
<td>stream-about, stream-all</td>
</tr>
<tr>
<td>Basic effector performatives</td>
<td>achieve, unachieve</td>
</tr>
<tr>
<td>Generator performatives</td>
<td>standby, ready, next, rest, discard, generator</td>
</tr>
<tr>
<td>Capability definition performatives</td>
<td>advertise</td>
</tr>
<tr>
<td>Notification performatives</td>
<td>subscribe, monitor</td>
</tr>
<tr>
<td>Networking performatives</td>
<td>register, unregister, forward, broadcast, pipe, break</td>
</tr>
<tr>
<td>Facilitation performatives</td>
<td>broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all</td>
</tr>
</tbody>
</table>

Table 1 - KQML Performatives (abstracted from EIWG, 1994)

The KQML language can be viewed as consisting of three layers: the content, message and communication layers, as shown in Figure 2.

The content layer specifies the actual content of the message, and the KQML standard itself has nothing to say about this. A KQML-conforming message could contain Prolog code, C code, natural language expressions, or whatever, as long as it is ASCII-representable. The set of performatives provided by the language, and shown in Table 1, constitute the message layer, which in turn forms the core of the language. This layer of abstraction provides the performative and specifies the protocol for delivering the message that subsumes the content.
By protocol, we mean the rules that agents must use when initiating and maintaining an exchange. KQML specifies several protocols, including synchronous (where a blocking query waits for an expected reply) and asynchronous (which involves non-blocking messages).

The chosen performative specifies that the content is a query, an assertion or any of the other category of performatives. The set of KQML performatives is extensible, but EIWG has tried to strike a balance between providing a small set (therby requiring overloading at the content layer), and providing a large set (where they are likely to overlap with one another, or else the distinctions between them become very fine). These performatives are neither necessary nor sufficient for all agent applications; however, agent application developers are encouraged to use them as specified in order to increase inter-operability across applications. If some application does not support all performatives, then it would not be KQML-compliant.

The communication layer encodes low level communication parameters, such as the identities of the sender and the recipient, and unique identifiers for the particular speech act. Note that the provision of a secure and reliable communications medium (essentially, the lower five layers of the seven-layer OSI model) tends to be taken for granted in most agent-related research. Only the upper two layers (presentation and application) are considered an issue.

For a flavour of KQML, here is an example of an instantiated performative.

{(tell
 : content "cost(bt, service-4, £5677)"
 : language standard prolog
 : ontology bt-services-domain
 : in-reply-to quote service-4
 : receiver customer-2
 : sender bt-customer-services)

The KQML performative in this message is tell, and the agent that is sending it seeks to inform some customer, customer-2, of a quote for performing some service, service-4, in reply to an earlier request from customer-2, to BT customer services. The content of the message is expressed in standard Prolog, and the ontology for BT’s services domain is assumed.

KQML has a number of shortcomings. Cohen & Levesque (1995) identify three general difficulties with it. First, the actual specification of KQML is ambiguous and vague. KQML does not in fact have a precise, formal semantics, as is normal of programming languages. Cohen & Levesque believe this is a real problem:

"Without [a precise semantics], agent designers cannot be certain that the interpretation they are giving to a “performativ” is in fact the same as the one some other designer intended it to have. Moreover, the lack of a semantics for communication acts leads to a number of confusions in the set of reserved “performatives” supplied” (Cohen & Levesque, 1995, p.65).

More troubling, it is not at all clear how one could go about giving the language a precise semantics, in such a way that it is possible to determine whether any given application is or is not KQML-conformant. And if it impossible to determine, accurately and unambiguously, whether or not a system that claims to conform to some standard actually does conform to the standard, one must ask what the point of the so-called standard is. Cohen & Levesque also argue that KQML suffers from mis-identified and missing performatives respectively. That is, they suggest that some KQML performatives are not what they claim to be, and that, in
addition, KQML is missing an entire class of performatives (commisives, which commit the utterer to a course of action).

Mayfield et al. (1996) also attempt an evaluation of KQML, and highlight some of its other merits and demerits. On the positive side, many prototype applications have already been constructed in North America and Europe that are KQML-compliant. It may be that, in the absence of any serious competition, KQML is becoming the de facto ACL standard.

2.2 Ad hoc ACLs

To date, many agent-based applications with collaborative agents use an ad hoc set of performatives within ad hoc agent communication languages. Many others, strictly speaking, do not have explicit ACLs; they communicate by depositing information in some shared data structure. Applications with their own ACLs are mostly speech act-based. They possess a limited subset of performatives, similar to KQML’s, but they are usually specified differently to their KQML equivalents, and they have different protocols. For example, SRI’s Open Agent Architecture (Cohen et al., 1994) has an ACL called the Inter-agent Communication Language (ICL). It possesses only three speech act types: solve (i.e., a question), do (a request) and post (an assertion to a shared data structure).

Naturally, the shortcoming of such ad hoc approaches is that it makes it non-trivial, if not impossible, for interoperation to occur between agent applications built by different developers. The case for having an ACL standard like KQML therefore seems compelling.

3 Languages for Constructing Agent Applications

There are many languages currently available to prototype agent-based applications. However, these do not warrant them be referred to as agent languages as many do. Wooldridge & Jennings (1995) write

“by an agent language, we mean a system that allows one to program hardware or software computer systems in terms of some of the concepts developed by agent theorists. At the very least, we expect such a language to include some structure corresponding to an agent”

p. 24.

This is fine, but it begs the polemical question “what is agent”? As there is no consensus yet (indeed, there may never be) with respect to this question, it is axiomatic that a language which some researchers may refer to as an agent language will not be referred to as such by others. What is true is that there are a whole range of languages which lend themselves to varying degrees to different types of agent applications and definitions. We have argued in Nwana (1996) and Nwana & Ndumu (1996) that the word “agent” is really a banner word for a heterogeneous body of on-going research and development, and we proceeded to provide a typology for software agents. We now use this typology to classify the languages we cover in this section. The typology includes the following agent types: collaborative, interface, mobile, information, reactive. Refer to either of Nwana (1996) or Nwana & Ndumu (1996) for more details.
3.1 Some languages that facilitate the building of agent applications

<table>
<thead>
<tr>
<th>Agent type(s)</th>
<th>Language Class</th>
<th>Example(s)</th>
<th>Major reference(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collaborative agents</td>
<td>Actor Languages</td>
<td>Actors</td>
<td>Agha (1986)</td>
</tr>
<tr>
<td></td>
<td>Agent-oriented programming languages</td>
<td>Agent-0, Placa</td>
<td>Shoham (1993), Thomas (1995)</td>
</tr>
<tr>
<td>Interface agents</td>
<td>Scripting languages</td>
<td>TCL/Tk, Safe-TCL, Safe-Tk</td>
<td>Oustershout (1994)</td>
</tr>
<tr>
<td>Information agents</td>
<td></td>
<td>Java</td>
<td><a href="http://java.sun.com/">http://java.sun.com/</a></td>
</tr>
<tr>
<td>Mobile agents</td>
<td></td>
<td>Telescript</td>
<td><a href="http://www.genmagic.com/">http://www.genmagic.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Active web tools</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Python</td>
<td><a href="http://www.python.org">http://www.python.org</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scheme-48</td>
<td><a href="http://photo.net/~jar/s48.html">http://photo.net/~jar/s48.html</a></td>
</tr>
<tr>
<td>Reactive agents</td>
<td>Reactive languages</td>
<td>RTA/ABLE</td>
<td>Wavish &amp; Graham (1996)</td>
</tr>
</tbody>
</table>

Table 2 - Some languages for developing different agent-based applications

Table 2 is by no means exhaustive; moreover, a significant number of the languages in this table are research demonstrators only. Also, note that Table 2 does not show many languages as suitable for developing collaborative (multi-)agent systems. This is because, in general, collaborative agents are much more complex than interface, information or mobile agents. They tend to require not only languages for their development, but architectures or platforms, e.g., DVMT (Durfee et al., 1987) or ARCHON (Jennings et al., 1995). Alternatively, they are constructed “from scratch” using some third generation language like C++, Smalltalk, or Prolog. Because of the length restrictions on this paper, we do not discuss any of the languages from Table 2 in any detail.

3.2 Traditional Languages are still used to construct agent applications

It is perfectly possible to implement, say, an Ada compiler in machine code. But we would not generally choose to use machine code for such a task, because there are better tools for the job: high-level languages like Pascal or C. In just the same way, it is possible to implement agent-based systems in languages like Pascal, C, Lisp, or Prolog. But as a rule, one would not choose to do so because such languages are not particularly well-suited to the job. A poor
choice of language will necessitate the re-implementation of what other languages offer for free. For example, it is probably not very wise to develop mobile agent applications in standard C, which allows the user to write and manipulate memory with impunity. Many people would agree that mobile agents could be malicious, and allowing arbitrary, erred access to memory is a sure way to allow trouble. Hence, writing such an application in C would require the implementation of a “safe” layer on top of C, which disallows direct memory access. But doing this will involve replicating the functionality that you get with the scripting language Safe-TCL. In summary, it makes sense to use the “right” (i.e., minimum cost, whatever the definition of “cost” is) tool for the job. Table 2 provides a “first-cut” set of several heuristics for making such choices.

Typically, object-oriented languages such as Smalltalk, Java, or C++ lend themselves more easily for the construction of agent systems. This is because the concept of an “agent” is not too distant from that of an “object”: agents share some properties with objects such as encapsulation, and frequently, inheritance and message passing. However, agents differ distinctly from objects vis-à-vis polymorphism. Objects respond to messages by invoking certain functions within them: polymorphism ensures that they do not necessarily “understand” the same messages in the same way. In contrast, agents must have a common ACL that they all understand.

4 Ontologies for Agent Applications

Any specific agent application is grounded in some domain ontology or ontologies. By ontology, we refer to the

"physical study of what exists. In the AI context, ontology is concerned with which categories we can usefully quantify over and how those categories relate to each other”


In other words, it is the foundational knowledge that agents need to share enough to communicate meaningfully. Consider the scenario, borrowed from Guha & Lenat (1994), of a teacher walking into a physics class. She presumes that at least the following are shared between herself and the students:

- most of the important vocabulary she will use (e.g., time, space, causality, friction);
- most of the knowledge involving these terms in the vocabulary.

These constitute the required shared ontology between herself and her students for any meaningful instruction to proceed. In other specific contexts, we (as humans) also bring different ontologies to bear, e.g., when we go into restaurants; in this case the menu lists a number of important additions to the required ontology.

Similarly, for computational agents, the common ontology contains the terms that will be used in communication between agents, and the knowledge relating to these terms. This knowledge includes their definitions, their attributes, and relationships between terms and constraints. Such a shared ontology is a sine qua non for any useful agent to agent communication using an ACL like KQML, because without it, the ACL is just syntax: much of the semantics derives from the domain ontology.

We classify work on ontologies in the context of agents in three different categories: ad hoc, “standard”, and global ontologies.
4.1 Ad hoc Ontologies

As with *ad hoc* ACLs, most current agent-based prototypes have some implicitly defined ontology. It is implicit in the sense that it is imposed by the designer “from the outside” - i.e., an exo-strategy is employed (Guha & Lenat, 1994). Therefore, an agent application in the domain of business process management, e.g., BT’s ADEPT demonstrator (Wiegand & O’Brien, 1996), builds the vocabulary of the business process domain into the task structures of the individual agents from the outside. Furthermore, the knowledge involving these terms is usually linked, implicitly, with the purpose (i.e., the task structures) and the co-ordination mechanisms of the agents in the demonstrator.

Naturally, there are problems with such an approach:

- Every demonstrator or application defines their own limited ontologies for their limited applications. There is little or no possibility for agent-based demonstrators based on the same domain, say business process management, to inter-operate - even if they used a standard ACL. This is because the ontologies of the two different systems would almost certainly be different.

- The approach is not scaleable. For example, if the prototype is required to be extended beyond the business process management domain, say to include the domain of network management, it would require a total reconstruction of the original demonstrator in order to accommodate the two sets of ontologies.

4.2 “Standard” Ontologies

To counter the sort of limitations mentioned in the preceding section, there are currently efforts to develop ontologies that can be shared across disparate software developers. Perhaps the best known, most advanced work in this area is the ARPA knowledge-sharing effort, (KSE), which includes the ontology sharing project. We shall here focus on this work.

The ARPA KSE program has argued for a while that a common ontology is a requirement if two or more heterogeneous agents or knowledge-based systems are to inter-operate by communicating information. An ontology is required in order to counter the lack of consistency between separate knowledge bases, in terms of their different vocabulary, underlying assumptions and, most importantly, the problem of semantics.

**KIF**

KIF (Genesereth & Fikes, 1992), an acronym for Knowledge Interchange Format, is an ongoing effort to address such issues as the latter. Essentially, it provides an *interlingua* for knowledge bases to inter-operate. In other words, for two agents with different legacy knowledge bases to inter-operate, both knowledge bases could be translated into KIF that will be the shared representation language. Given $n$ knowledge base formats, the use of an interlingua such as KIF necessitates at most $2n$ format converters, as opposed to the $n(n-1)$ converters that would otherwise be required.

KIF is essentially the first-order predicate calculus, recast into a Lisp-like notation. The following example depicts the content slot of an *untell* KQML performative expressed in KIF; this example is borrowed from Fininet *et al.* (1994).

```plaintext
(untell
  :language KIF
  :ontology motors
)```
The content slot of this message says that the torque of object motor1 at simulation time (sim-time) 5 is 12 kgf. It is assumed that the ontology motors (mentioned in the ontology slot of the message) defines the terms torque, sim-time, kgf and so on. Note that KIF is not an implemented knowledge representation language. Rather, it is an implementation-independent interlingua that allows for precise and unambiguous knowledge representation. For more details, see Genesereth & Fikes (1992).

**Ontolingua**

Ontolingua is another ARPA-sponsored effort towards reusable ontologies. Figure 3 depicts its mode of operation.

![Ontolingua's mode of usage](http://www-ksl.stanford.edu/knowledge-sharing/ontolingua/)

Thus Ontolingua is a domain-independent translation tool, which has nothing to do with the intellectual task of defining ontologies. Like KIF, it is also not an implemented language; rather, it allows for the “explicit specification of a conceptualisation” (page 199), which is what Gruber defines an ontology as.

Ontolingua is a declarative and formal language that can be used to capture and represent ontologies at the knowledge level (Newell, 1982), but which acts as a mediating representation for translating from “off the shelf” ontologies into several bespoke symbol level knowledge representation (KR) languages, e.g., Loom (MacGregor, 1991).

The knowledge sharing arises because the same Ontolingua specification, say some generic knowledge-based planning program, can be translated into different symbol-level languages. The ontology of such a planner would include descriptions such as objects, events, resources, constraints, etc., and the planner assigns resources and times to objects and events (Gruber, 1993). Reuse occurs when the same ontology is reused in different applications or by different developers.

Ontolingua statements and axioms are written in an extended KIF notation and natural language sentences, though the latter are not parsed. The Ontolingua system includes a KIF parser and syntax checker, a consistency checker, a hypertext editor and several translators from KIF to other bespoke languages. For more details, consult Gruber (1993) and Ontolingua's web page.²

4.3 Global Ontologies

There are also several efforts aimed at defining and generating more global ontologies. By this, we mean that rather than being directed at specific domains, global ontologies are intended to be in some sense general. Such ontologies may be provide invaluable substrates for agent-based applications. Two notables ones are the WordNet and the Cyc projects. Theses are discussed briefly below.

WordNet (Miller, 1995) is an on-line lexical database with more than 166000 word form and sense pairs. For example, the word form “back” is interpreted as a noun in some linguistic context, as a verb in another, and even as an adjective or adverb in yet others. Each of these contexts are entered separately into WordNet. WordNet also includes many semantic relationships between words and word senses, and it also contains several semantic relations including synonymy, antonymy, hyponymy, meronymy, troponymy and entailment. It must be emphasised that WordNet is not really designed for agent applications; clearly, it is of much relevance to natural language understanding applications. However, it is possible to use it to complement further the ontology defined for some agent application.

Cyc (Guha & Lenat, 1994; Lenat, 1995) is a project that began in 1984 with the ambitious goal of capturing and representing commonsense knowledge. The original motivation for Cyc was to address the brittleness problem of expert systems. In brief, the point is that while a medical expert system might in some sense be an expert in (say) blood diseases, it will typically be unable to address any questions outside this narrow domain. Cyc’s inventors believe that the way to solve this problem is to build a system that has broad (though not necessarily deep) knowledge: the kind of knowledge that would be required by anyone attempting to understand an encyclopaedia article. Examples of such knowledge are:

- You have to be awake to eat.
- You can usually see people’s noses but not their hearts.
- You cannot remember events that have not happened yet.
- If you cut a lump of peanut butter in half, each half is also a lump of peanut butter; but if you cut a table in half, neither half is a table.

Cyc currently contains $10^5$ concepts, and $10^6$ commonsense axioms that have been hand-crafted into it; millions more have been inferred and cached by Cyc (Lenat, 1995). Today, Cyc’s inventors believe that the first real applications of Cyc will be into mainstream computing, particularly in the domain of information management (Guha & Lenat, 1994). Lenat (1995) posits that Cyc could form the standard ontology underlying the world wide web (WWW) and electronic commerce. In such contexts, Cyc-enabled applications which may (or may not) be agent-based, could facilitate interoperability. If the Cyc dream comes true, it will obviate the need for most knowledge representation languages and current efforts including KIF and Ontolingua. For example, the content of a KQML expressions could be specified in CycL, the language in which Cyc’s concepts and axioms is encoded. It is unlikely that such a global ontology as Cyc will be needed or available (it is currently too costly) for relatively small-scale agent demonstrators and applications in the short-term. However, if Cyc even approaches the functionality predicted by its inventors, then it will revolutionise not just agent applications, but much of computing beyond.
5 Support Computing Technologies

In tandem with some developments in languages, ontologies, and protocols, there are also several developments in the supporting computing technologies that facilitate the design and implementation of agent applications. Briefly, we highlight some of these below. However, to get a better appreciation of this section, perhaps a brief historical diversion is required.

Up until the mid-1980s, computing was dominated by big, monolithic mainframe applications. Such applications still exist in many large organisations. The costs that have been invested in such systems over the years, and their criticality to the smooth and everyday functioning of the organisation ensure that their replacement - no matter how persuasive the arguments for such moves are - is anathema to the senior management of these organisations. This is the legacy software problem and, literally, it gets worse by the day.

The idea of client-server computing was proposed as an alternative to the centralised computing model more than two decades ago. Since then, client-server computing has prompted a Kuhnian paradigm shift in the computing industry. However, the idea only began to really change computing in the 1980s, with the development of better hardware, and the demand for applications that matched this hardware. The new hardware included personal computers (PCs) and local area networks (LANs) such as Ethernet. Client-server computing changed computing by replacing monolithic mainframe applications, which were typically accessed via green-screen terminals attached directly to these mainframes. Today, clients, which are typically PCs with sophisticated graphical user interfaces (GUIs), interact with server programs that manage shared resources; server applications typically manage databases.

5.1 Agents and Client-server Computing

The key point here is that the client-server model naturally lends itself to the implementation of software agent systems. To see why, first note that there are several motivations for having multiple agent systems. They include (Nwana, 1996):

- To solve problems that are too large for a centralised single agent to do due to resource limitations or the sheer risk of having one centralised system;
- To allow for the interconnecting and interoperation of multiple existing legacy systems such as expert systems and decision support systems;
- To provide solutions to inherently distributed problems such as distributed sensor networks (cf. Durfee et al., 1987) or air-traffic control;
- To enhance modularity (which reduces complexity), speed (due to parallelism), reliability (due to redundancy), flexibility (i.e., new tasks are composed more easily from the more modular organisation) and reusability at the knowledge level (hence shareability of resources).

These points all relate to the merits of decentralised computing over monolithic systems: agent-based systems are decentralised systems. The client-server model thus closely matches the decentralisation associated with agent technology. To summarise, we believe there is a synergistic relationship between agent technology and client-server computing, and hence that client-server architectures provide a natural environment within which to develop agent applications.
5.2 Agents and Distributed Objects

Although we are yet to see the full impact of the first client-server revolution, advances in hardware are already fuelling a second client-server revolution (Orfali et al., 1996). This second revolution is being driven by wide area networks (WANs) and object-oriented technology. The integration of these two relatively old technologies yields a powerful new approach for achieving large-scale client-server computing: distributed objects. As Orfali et al. (1996) write:

“...by and large, today's client-server applications remain difficult to build, manage, and extend. Distributed objects change this. With the proper packaging and infrastructure, objects will help subdivide today's monolithic client-server applications into self-managing components that can play together and roam across networks and operating systems. Component-like objects allow us to create client-server systems by assembling “live blobs of intelligence and data” in an infinite number of lego-like arrangements. These components represent the ultimate form of client-server distribution and prepare us for the near future when millions of machines - mostly desktops - will be both clients and servers”

p. 15.

Distributed objects allow for the packaging of software into components with well-defined interfaces which, in turn, offers other advantages. First, new information systems can be built by assembling various components, lego-like, as long as the interfaces are consistent with one another. Secondly, components or objects are portable: they can run on different operating systems such as OS/2, Macintosh or UNIX. In 1989, a consortium of object vendors grouped together to form the Object Management Group (OMG).

![Diagram of the OMG Object Management Architecture](image)
The key to CORBA is its Interface Definition Language (IDL). The IDL is a declarative language, which allows for the definition of Application Program Interfaces (APIs). The IDL definition language allows for the definition of the objects, their attributes, the methods they export and the parameters of these methods. IDL has no control structures or variables. IDL, together with the ORB, constitute the object bus.

Objects written in different programming languages, packaged as binaries, and running on different operating systems can have IDL interfaces so that they can inter-operate, and invoke each others’ methods. The point of this all is that objects written in (say) Cobol, Lisp, or C++ linked to ORB can now inter-operate in various client-server modes across different operating systems. So clients do not need to care about where the distributed objects reside or what operating systems they are running on. This is a significant development, even within client-server computing. For example, it would allow workflow objects on some server in Amsterdam to inter-operate with some customer’s business process objects in Ipswich. Therefore, a fourth advantage of distributed objects is that new objects may co-exist with legacy applications, if the latter are provided with IDL interfaces. OMG’s ultimate goal is to create a collaborative client/server component-based environment.

What has all these got do with agents? Simply that such emerging technologies and standards ensure that computing will continue to be increasingly decentralised. Such decentralised architectures, as noted earlier, provide natural substrates for the construction of an agent layer. This agent layer will provide many of the core services required to build agent applications.

6 Discussion & Conclusion

In this closing section, we briefly discuss some of the issues that have emerged in our survey. We begin by noting that at several points, we found ourselves discussing standards of one sort or another. Standards will be important in agent-based systems, for exactly the same reason that they are important in distributed systems and telecommunications generally: agents need to be able to talk to, and understand one-another. Yet while the importance of standards seems clear, there are also a number of dangers associated with standardisation. It is worth pausing to consider what these dangers are, and how they might adversely affect the development of agent systems.

First, there is the danger of standardising too early, or, equivalently, of picking the wrong standard. Arguably, standards are most successful in stable domains, or domains where the future development of the technology is easy and safe to predict. This is not the case with the agent field. To see why, consider the world-wide web. This enormously popular service did not exist before 1993; nor could its enormous success have been predicted. If anyone attempted to standardise information services for wide area networks in 1992, they were almost certainly wasting their time. Unfortunately, the agent field is developing at a similarly rapid pace. This makes standardisation a particularly difficult problem for agent researchers. It could be argued that KQML, (the agent communication language discussed in detail above), is an attempt to standardise too early, as the requirements for ACLs are not sufficiently understood: there are too few real multi-agent systems in existence to know what features an ACL must support.

Just as it is possible to standardise too early, so it is possible to standardise too late. A good example here is the X.400 email standard, which was widely expected to take over from the internet’s Simple Mail Transfer Protocol (SMTP). This did not happen, despite the technical
superiority of X.400 over the cruder (and comparatively ancient) SMTP standard. One reason for this failure was simply that by the time X.400 had passed through the time consuming international standardisation process, and working X.400 systems became available, SMTP was already established as the de facto standard. So, although X.400 was technically right (in that it provided for many sophisticated and powerful services) it arrived too late to have the impact it perhaps deserved. A similar situation within the agent field may be occurring with Telescript (White, 1994), a programming language and development environment for mobile agent applications. Telescript was eagerly awaited within the Internet agent community, but by the time it was made available to the public, the Java programming language had already been released, and was supported within the popular Netscape world-wide web browser. As a result, Java had a very large end-user base, making it much more attractive to software developers than Telescript. Now Java does not provide facilities for building mobile agent applications in the way that Telescript does. In this sense, the two languages are not really competing - but Java appears to be winning the race anyway.

Standards will be essential if agent technology is to realise its potential. But, as we hope to have made clear, standardisation is as much about timing and judgement as it is about technical issues.

To sum up, we currently have no firm convictions about exactly what form future agent based systems will take. But we are convinced that agents have much to offer. In this article, we hope to have illustrated some of the technologies that are likely to underpin this promising new software industry.

7 Acknowledgements

We acknowledge discussions with Divine Ndumu and Mark Wiegand that have shaped some of the ideas propounded in this paper.

8 References


