

## CHAPTER 4 ARCHITECTURAL PERSPECTIVES

### 4.1 Introduction

This chapter describes architectures of intelligent assistant systems. A number of systems and research projects which fall under the heading of ‘intelligent systems’ and which are related to the concept of an intelligent assistant which has been proposed, are examined in conjunction with general aspects of software architecture. This chapter also describes an investigation into the trend towards distributed client-server platform-independent systems and its implications for the development of an architecture for the proposed intelligent assistant system.

### 4.2 Software Architectures

Recently, software architecture has emerged as an important field of study for software engineering researchers and practitioners [Bass et al., 98]. Software architecture can be defined as [Shaw and Garlan, 96];

*“Structural issues concerning the organisation of a system as a composition of components; global control structures; the protocols for communication, synchronization and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives”.*

Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Good architectural design has always been a major factor in determining the success of a software system. However, while there are many useful architectural paradigms

(such as pipelines, layered systems, client-server), they are typically understood only in a pragmatic way and applied in a non standard fashion. Consequently, software systems designers have been unable to exploit commonalities in system architectures, make principled choices among design alternatives or specialize general paradigms to specific domains.

A sound basis for software architecture promises benefits for both development and maintenance. For development, it is increasingly clear that effective software engineers require proficiency in architectural software design. First, it is important to be able to recognise common paradigms, so that high level relationships among systems can be understood and so new systems can be built as variations of old systems. Second, getting the right architecture is often crucial to the success of a software systems design. Third, detailed understanding of software architectures allows the engineer to make principled choices among design alternatives. Fourth, an architectural system representation is often essential to the analysis and description of the high level properties of a complex system. Fifth, fluency in the use of notations for describing architectural paradigms allows the software designer to communicate new system designs to others.

One of the difficulties in working with software architecture is that different designers may interpret an architectural paradigm in different ways. For example, although two designers may both claim that their design is built around a client-server paradigm, they may mean quite different things by that term [Berson, 92]. A related problem is that several systems may be designed with similar architectural structures, but the designers never recognise that the similarities exist, consequently they overlook opportunities to capitalise on the experience of other designers.

Choosing the most appropriate architecture for a given situation remains an open problem [Bass et al., 98]. The rules of the style usually determine how to package components, for example, as procedures, objects, or filters. As a result, components cannot usually be interchanged across styles, for example code, may not be reusable because its interface makes it incompatible.

## **4.3 Intelligent System Architectures**

Given the above discussion, it follows that in order to fully understand the architectural requirements for an intelligent assistant system, an examination of existing architectures must take place. Examining these architectures in conjunction with the characteristics of the problem domain will lead to an improved understanding of the nature of architectures for intelligent assistant systems and assist with the specification of an architecture for the proposed system.

In the following sections, three assistant systems will be analysed from an architectural perspective. These systems all operate in the area of management decision making and have diverse architectures which are based around the agent-orientated paradigm. For each system a brief description of its origin will be given, followed by an examination of its architecture and implementation as well as a discussion about its impact on the decision makers who use the system.

### **4.3.1 ADEPT**

The ADEPT (Advanced Decision Environment for Process Tasks) project [Alty et al., 94] had as its main aim the implementation of an agent-based approach to managing business processes. The system was expected to:

- Allow decision makers access to relevant information wherever it is situated and request and obtain information from other departments within the organisation and outside the organisation.
- Provide timely and relevant information to decision makers which may not have been asked for.
- Inform decision makers of changes made elsewhere which impinge on current decision processes.
- Identify parties which may be interested in the outcome and results of the decision making activity.

The ADEPT system involves the transformation of some business process descriptions into a number of agencies, each with a connection to some common communication medium. An agency is recursively defined: an agency consists of a single responsible (or controlling) agent, a set of tasks the agent can perform and a possible set of sub-agencies. This relationship between sub-agency and responsible agent can be viewed as a type of social commitment and provides a mechanism for the encapsulation and abstraction of services. The responsible agent provides a specification of the services that it can and is willing to provide to its peers, even though some of the activity will require the assistance of its sub-agents.

Essentially each agent is able to perform one or more 'services', where a service is a unit of problem solving activity. The simplest service (called a task) represents an atomic unit of problem solving endeavor. These atomic units can be combined to form 'complex services' by addition of ordering constraints (e.g. two tasks can run in parallel, must run in parallel, or must run in sequence) and conditional control. The nesting of services can be arbitrarily complex and at the topmost level the entire business process can be viewed as a service.

Services are associated with one or more agents which are responsible for managing and executing them. Each service is managed by one agent, although it may involve execution of sub-services by a number of other agents. Since agents are autonomous, there are no control dependencies between them; therefore, if an agent requires a service which is managed by another agent, it cannot simply instruct it to start the service. Rather, the agents must come to an agreement about the terms and conditions under which the service will be performed (called a service level agreement).

In ADEPT, all agents have the same architecture (figure 4.1) which involves a 'responsible agent' that interacts with its peers and the 'subsidiary agencies' and tasks within its agency. An agents agency represents its domain problem solving resources. The responsible agent has a number of functional components concerned with each of its main activities - communication, service execution, situation assessment and interaction management.

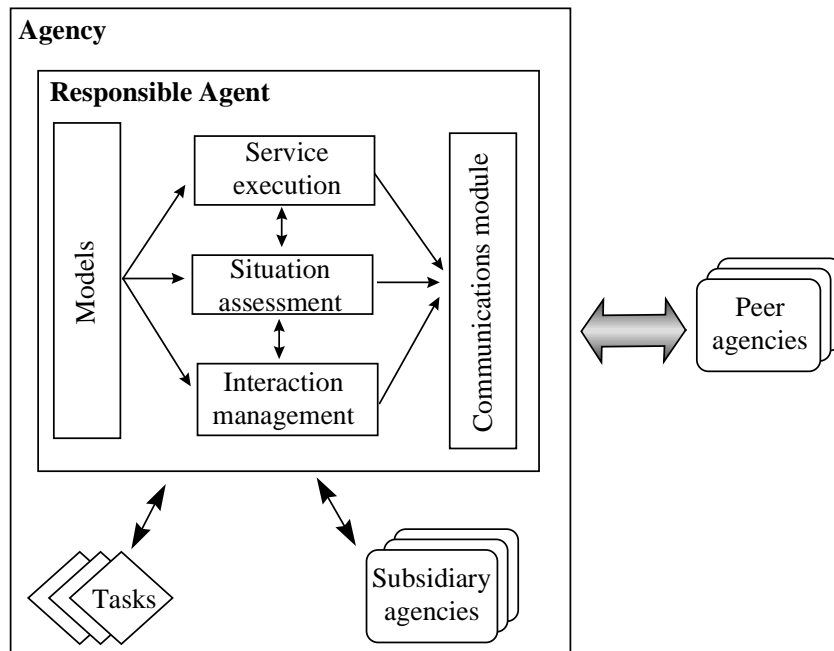


Figure 4.1 - ADEPT Agent Architecture

- The **communications module** routes messages between an agent and its agency and between peer agents.
- The **interaction management module** (IMM) provisions services through negotiation. It generates initial proposals, evaluates incoming proposals, produces counterproposals and finally accepts or rejects proposals. If a proposal is accepted then it generates a new SLA (service level agreement).
- The **situation assessment module** (SAM) is responsible for assessing and monitoring the agents ability to meet the SLAs it has and the potential SLAs which it may agree in the future. For example, if a service is delayed then the SAM may decide to reschedule it, to renegotiate its SLA, or to terminate it altogether.
- The **service execution module** is responsible for managing services throughout their execution. This involves execution management, information management and exception handling.
- The **models** are the primary storage site for SLAs to which agents are committed, descriptions of the services the agent can provide and generic domain information.

The agent knowledge base is represented as a set of strategies and a mechanism for selecting between them, which is implemented using a modified version of the CLIPS system. For agents to participate in the ADEPT environment, it is necessary for them to communicate using a common expressive language. This common language KIF [Genesereth and Fikes, 92], consists of a protocol and a syntax for expressing information and allows agents to interpret other agents intentions.

The overall ADEPT framework contains three basic components (or layers): The application layer is at the level of user interaction, where service descriptions and other information is gathered. The management layer contains all the agents and their sub-agents and the convergence layer presents a standard interface between the agent system and the underlying platform. The agents themselves are implemented on top of a CORBA compliant distributed platform.

The ADEPT system was tested by British Telecom, who used the system to support a number of concurrent business processes, such as the generation of customer quotations for designing a network to provide particular services to a customer. This would involve up to six parties: the sales department, the customer service division, the legal department, the design division, the surveyor department and the provider of an out-sourced service for vetting customers. For representing this business process, a full description of the process was translated into an agent system, where (at an abstract level) each agent represented a distinct department involved in the process.

ADEPT agents themselves do not carry out the totality of a business process; much of the work is ultimately carried out by humans or other software (often legacy) systems that are externally interfaced to ADEPT agents. ADEPT agents do need a certain amount of domain knowledge (or meta knowledge) and in most cases this domain knowledge can be of considerable size and complexity. However, this type of information can be extracted during a business process re-engineering exercise, which many efficient organisations conduct.

### **4.3.2 ARCHON**

The ARCHON (ARchitecture for Cooperative Heterogeneous ON-line systems) project [Jennings et al., 96a] had as its aim the development of a general purpose architecture which would allow pre-existing expert systems, dealing with different aspects of decision making of a given complex environment, to cooperate in a mutually beneficial way. The main design feature of ARCHON was that it put an emphasis on loose coupling as a means to increased cooperation between a set of systems. Thus, systems that participate in mutual cooperation are conceived of as autonomous and capable of completing their allocated tasks without much reliance on other systems in the community, but benefiting from each others activities through cooperation mechanism.

ARCHONs design objectives [Wittig et al., 94] were for the interworking of semi-autonomous agents. It can complement integration architectures that provide for tight coupling of systems, such as client-server architectures in which a client would demand a service and a server is mandated to provide that service. ARCHON agents may enter into a client-server relationship with each other for a contracted set of tasks, but are never designated (pre-destined) to perform one or the other of those roles. ARCHON agents can pass unsolicited information to their acquaintances, leaving it to the recipient to decide what to do with it.

Figure 4.2 illustrates the modules which form the ARCHON architecture and shows the interface to the intelligent system.

- The architecture needs a communications facility, which is called the **High Level Communications Module (HLCM)**. It is the ‘high level’ since it not only provides communications facilities, but also addressing and filtering. For example, if the domain system produces a result that may be relevant for other agents, the Planning and Coordination Module (PCM) just asks the HLCM to send it to all interested agents without specifying them.
- The **Agent Information Module (AIM)** provides an object orientated information management model and a query language to define and manipulate the information.

- The **Agent Acquaintance Models** (AAM) contain representations of other agents in the community in terms of their skill, interests, current status of workload, etc. Agents will not actually maintain models of all agents in the community, simply a subset based on similar interests/capabilities.
- The **Self Model** (SM) is an abstract representation of an agents domain system. It primarily contains information about the current state of this system i.e. its workload, or what tasks are being executed, but also embodies the precompiled plans (behaviors).
- The **Monitor** is responsible for the control of the intelligent system and for passing information to and from it.
- The **Planning and Coordination Module** (PCM) represents the main reflective part of ARCHON. If an exception occurs, it is the task of the PCM to reason about it and find a way out.

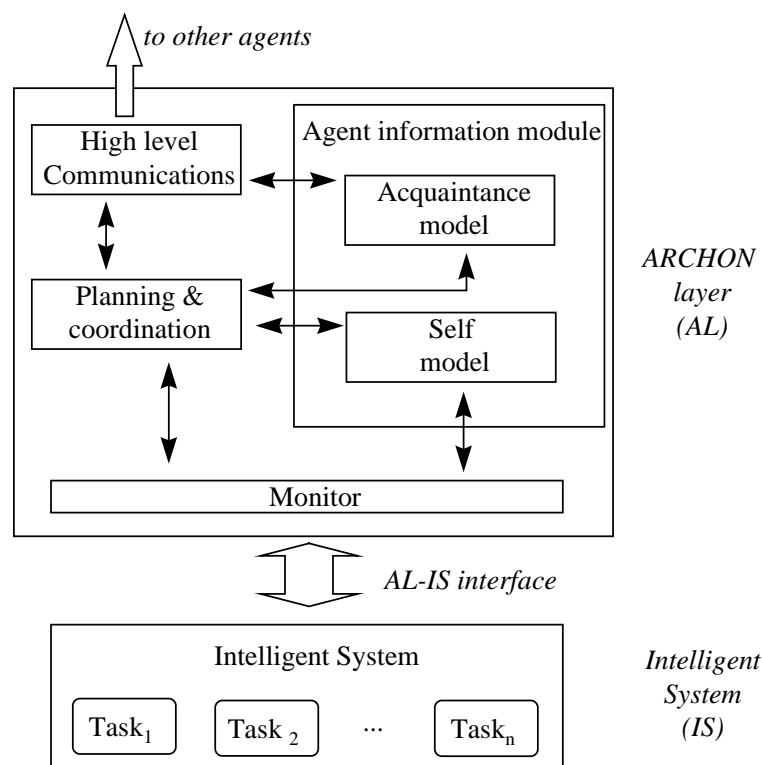


Figure 4.2 - ARCHON Agent Architecture

In an ARCHON agent community there is no centrally located authority, each agent controls its own IS (Intelligent System) and mediates its own interactions with other agents. The systems overall objectives are expressed in separate local goals of each

community member. Because the agents goals are often inter-related, social interactions are required to meet the global constraints and to provide the necessary services and information. Such interactions are controlled by the agents AL (ARCHON Layer), for example: asking for information from other agents, requesting processing services from them, or volunteering information to other agents. Essentially an agents AL needs to control tasks within its local IS and decide when to interact with other agents.

The ARCHON architecture concentrates upon loose coupling of semi-autonomous agents. If an organisation has a collection of pre-existing systems, each dealing with a separate aspect of the same domain, then this architecture presents an opportunity for bringing these together into a useful co-operative framework. However, ARCHON was not only designed for pre-existing systems, but for providing cooperation between any set of semi-autonomous systems, which had not been adequately explored. Before this can be achieved, one must consider that ARCHON, on its own, restricts an integration approach to formulating a solution only in terms of loosely coupled, semi-autonomous agents.

The ARCHON project's principal test environment was in the domain of alarm analysis in the electricity supply network of Iberdrola in Bilbao, Spain. The increased automation and complexity of the automatic controllers has brought the electricity utility to the point where human intervention is scarcely needed, but whenever it does occur, the responsibility on the decision maker is even greater than ever before. This increase in automation also produced an increase in the amount, reliability and complexity of information received. In order to help human operators during the monitoring of the network at Iberdrola, several expert systems were developed over the years, such as; alarm analysis system and black-out identifier system - which were modelled using ARCHON agents. When one agent (expert system) identified a problem (e.g. alarm message caused by a fault), it could analyse the situation and (voluntarily) inform other agents which it considered needed to know. This inter-agent communication reduced the need for the operator to input information from one system to another, allowed other systems (agents) to have more timely information

about a potential situation and cumulate more efficient and precise information communication to the human operator (decision maker).

Although several applications were evaluated using the ARCHON approach, all had some pre-existing systems. The project itself did not evaluate how the ARCHON approach might complement an existing client-server or distributed type integration environment. In addition, the ARCHON project has not been extended to systems with many (hundreds of) agents and it is likely that in such a situation the designer may need to consider if some of the smaller agents need to be coalesced again.

### **4.3.3 RISKMAN2**

The RISKMAN2 project [Moynihan et al., 94] aimed to develop a critiquing system to support risk analysis for software development projects. This project built upon the lessons learned from Integrated Management Process Workbench project [Jenkins et al., 87], which developed the RISKMAN tool [Verbruggen et al., 89].

The goal of the RISKMAN2 project was to build a tool which would enable project managers 'walk around' a proposed project and help them anticipate any major risks to which the project might be exposed. The major inputs to the system are 'Risk Drivers', where each risk driver is seen as contributing in a linear, additive fashion to the risk measure (i.e. value) for one or more areas of risk management. The values of the risk drivers are elicited from the user (a project manager) by a Project Definition Tool. The major output of the tool is a risk report, which is split in two sections. The first section provides the computed risk measures, one for each of the risk management areas identified. The second section, consists of advice paragraphs, where each paragraph consists of text offering comments and advice to the user relating to the management of risk on the project. The user is provided with the ability to query the risk report to determine the basis upon which the risk measures and the advice paragraphs were arrived at.

The outline architecture of RISKMAN2 [Power, 94][Henry, 94] is illustrated in figure 4.3 and contains five major components: Generic Project Model (GPM), the software

risk taxonomy, the Risk Analysis Daemon Library (a collection of risk analysis agents, each of which specialises in a different element of the risk taxonomy), the Blackboard and the Reporter. The user begins a session by instantiating the GPM for the project under consideration. Then the risk analysis daemons (agents) inspect the instantiated GPM, compute their results, and write these to the blackboard. In computing their results, daemons have access to results which other daemons may have written to the blackboard. When the daemons have completed their analysis, the user can invoke the reporter to retrieve the daemons results from the blackboard and display the results in different ways.

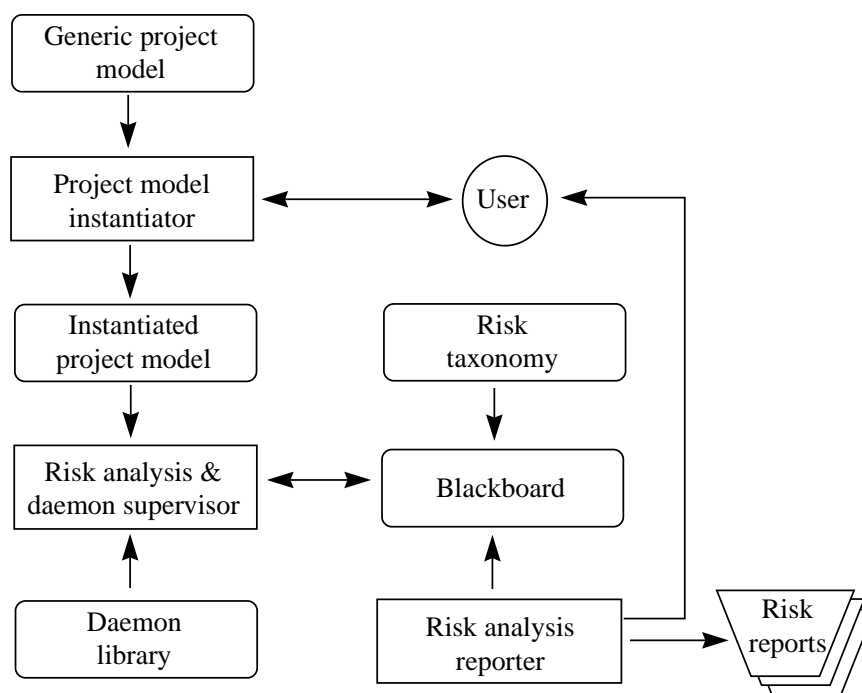


Figure 4.3 - RISKMAN2 Architecture

The RISKMAN2 components are:

- The **Generic Project Model** (GPM) is a very general, customisable model of a software project. The GPM takes the form of an object-model and uses the notation and semantics of the OMT modelling technique.
- The project **Risk Taxonomy** is a description of the types of risk to which a software development project can be exposed. For example, the risk of a product not meeting its performance requirement. The taxonomy takes the

form of a classification in which risk types are broken down into sub-types and is based on the US AIR Force risk management taxonomy [USAF, 88].

- The Generic **Project Model Instantiator** (PMI) through dialogue with the user, customises the GPM to produce a model of the particular project under consideration. The PMI is driven by the GPM in the sense that the PMIs goal is to instantiate as many of its elements as possible. The PMI does this by asking the project manager a series of multiple-choice questions, to get values for the GPM elements.
- The **Risk Analysis Daemons** are mini-experts in some aspect of risk management, one for each area in the risk taxonomy. Each daemon (or agent) is structured as a set of small rule-based production systems, with variables in the conditional rule-set corresponding to elements in the GPM.
- The **Daemon Library** acts as a storage point for a collection of daemons.
- The RISKMAN2 **Blackboard** is a generic blackboard. When a daemon has performed its risk calculations, it must write its results to the appropriate place on the blackboard. In this way every daemon can access the results of any previously executed daemon.
- The function of the **Risk Analyser Reporter** is to deliver a structured report to the user. It does this by accessing the conclusions written to the blackboard by the daemons.

All daemons are controlled by a ‘daemon supervisor’. A daemon begins its task by attempting to assign values to the variable in the conditional part of its rule-set. If it finds the data it needs, it proceeds with its analysis (executes its production rules). If it cannot find all the data it needs, the daemon terminates and returns ‘failure’ to the supervisor, in which case the supervisor will schedule that daemon to execute at a later time. Eventually, assuming data becomes available, the daemon writes its conclusions to the blackboard, from which they can be read by other daemons and the risk reporter. A daemon’s conclusions are in three parts: A **risk metric** which is a measure representing the degree of risk which the daemon considers is present in the project; **why text** which is hard-wired into the daemon and explains how the daemon

worked and reached its conclusions; and **advice text** which includes suggestions as to how the particular risk may be reduced or otherwise managed.

The RISKMAN2 architecture represents an open and flexible architecture which facilitates, over time, the ability to improve the system by adding extra daemons, enhancing the GPM, adding or replacing the risk taxonomy and the blackboard approach allows for a degree of modularity and dynamic control. However, a number of weaknesses exist in relation to the daemons; The daemons are organised like an army in the sense that each daemon must write to a single place on the blackboard and 'higher up' daemons must only take as their input the output of lower daemons in the taxonomy. Also, no mechanism exists for handling disagreement or conflict between daemons, or to ensure the system can still function if the user does not fully instantiate the GPM. In addition, the variables in the conditional part of a daemons rule-set are constrained to be elements of the GPM - in other words, a daemon may only view the project through the lense of a GPM. This excludes the possibility of building daemons which 'see' projects in a richer or different way.

#### **4.3.4 Review of Architectures**

In this section, the three different architectural approaches to implementing intelligent assistance systems for management decision making are contrasted to assess their comparative strengths and weaknesses in order to appraise the suitability of agent-based architectures to support management decision making and the possibility of adapting these approach to the domain of software project planning.

The ADEPT system encapsulates business processes in agents, where each agent can provide a 'service' (or task) to other agents and enters into 'service agreements' with other agents to provide a service to them. In the ADEPT hierarchy, agents at a high level may only enter into such service agreements with their peer agents, although these agents usually require the services of subordinate agents (those lower in the hierarchy) to provide such services. Such service provisioning requires complex inter-agent communication, as well as the ability for service negotiation and a knowledge of

the agent hierarchy. Therefore each agent has additional structures to cope with such communications, which represents an implementation overhead for each agent in addition to the overall system overhead of continuous complex inter-agent communication traffic. Further, all knowledge in the ADEPT system is embedded in 'models' within its agents, which does not allow for the dynamic updating of the ADEPT knowledge base - which may only be achieved by manually replacing agents. A positive aspect of the ADEPT implementation is the support for multiple platforms. Agents are implemented on top of a 'convergence layer' which represents a communications interface to the operating system. In the case of ADEPT, the CORBA broker DAIS was used in implementation however further expansion was envisaged (although not tested) using OLE (Object Link and Embedding - now ActiveX).

It is worth mentioning that members of the ADEPT project consortium considered the project successful and provided a useful step forward in investigating agent-based support systems [Alty, 97]. However, they recognized that the implementation of the user-orientated section of the tool (application layer), through which the user interacted with the system, was poorly constructed and diminished the end users productivity.

While there are many interesting architectural lessons to be learned from the ADEPT approach, it does not provide a suitable basis upon which to implement an intelligent assistant system for software project planning. The ADEPT approach to intelligent assistance is closely related to software process enactment environments [Finkelstein et al., 94] as the system simply automated existing (business) processes. ADEPT agents are not adaptable to a situation where there is no pre-defined process, nor can they provide a facility to define additional processes (such as a software project plan). In addition, much of the knowledge required by ADEPT agents is restricted to low-level design guidelines and ADEPT agents have an inherent lack of expressive power, with explicit support only for top-down design processes [Jennings et al., 96b].

The ARCHON system essentially provides an agent-based shell or wrapper within which a pre-existing expert system may operate and exchange information with other subsystems via an inter-agent communication mechanism. Unlike ADEPT agents,

ARCHON agents do not themselves directly contain any problem solving data, but contain a model of other agents in the system and the ability to communicate with them. Like ADEPT, all ARCHON agents must have structures to cope with complex communications, which represents an implementation overhead for each agent, in addition to the overall system overhead of continuous complex inter-agent communication traffic. It should be noted that in this agent community there is no centrally located authority - the system is simply made up of a community of loosely coupled agents, each of which represents a subsystem capable of completing some task. Further, the ARCHON approach was designed only with a small number of pre-existing systems in mind. To expand the ARCHON system requires the addition of new agents, however, these agents represent pre-existing subsystems which must be in place before the agent can be deployed - thus adding an extra layer of complexity and inhibiting the possibility of dynamically updating the system. Further, the ARCHON, unlike ADEPT implementation does not provide any support for multiple platforms as it is tied directly to the underlying operating system.

The ARCHON system does not directly provide a suitable framework for a software project planning intelligent assistant system. Overall, ARCHON can be characterised as approximating the functionality of a DSS, as it provides for a number of pre-existing systems to operate in a more timely manner and provide the user with more accurate and timely information upon which to base a decision. ARCHON agents themselves do not conduct any problem solving or interact with the user to solve their problems. ARCHON agents were designed to interact with other ARCHON agents and their associated intelligent systems and, as such, are not readily adaptable to directly represent an area of problem solving activity for a given domain (such as software project planning). However, the ARCHON approach could be used to coordinate multiple (discrete) intelligent systems for software project management where ARCHON agents represent a software project planning system, software project risk analysis system, etc.

The RISKMAN2 approach provides a framework in which a community of agents can operate and exchange information. Unlike ADEPT or ARCHON agents, RISKMAN2 agents use a simple form of communication, that of a blackboard structure, and in addition have a controlling entity which directs the agent community,

thus reducing the overheads associated with inter-agent communication. In addition, this approach allows agents to be pro-active, in that all agents can access the blackboard and may respond in an opportunistic manner, unlike ADEPT or ARCHON agents which rely on explicit message passing. Another advantage the RISKMAN2 approach has over the others is the ability to dynamically update the knowledge base via the 'Daemon Writer Kit' [Power, 94], which provides a template for the construction of agents. As every agent is autonomous and fully self-contained, any agent may be removed or updated without affecting the rest of the system. However the RISKMAN2 approach also has a number of problems: Agents are only capable of viewing a project through a predefined project model (the GPM) and agents may only take input from agents which are lower in the hierarchy than themselves. Like ARCHON, the RISKMAN2 implementation does not provide support for multiple platforms and is tied directly to the underlying operating system. Further, RISKMAN2 suffered from a number of implementation problems and, unlike ADEPT and ARCHON, the end system was not sufficiently tested in real world situations.

The RISKMAN2 approach has potential to be of use in implementing an intelligent assistant system for software project planning. Unlike both ADEPT and ARCHON, the RISKMAN2 system was not simply a DSS. It also incorporated features of both an expert system - in that it could diagnose the existence of a given situation (e.g. high level of risk in a certain area), and an expert critiquing system - in that it could provide advice on how to deal with that situation (e.g. advice on mitigating an identified risk in a particular area). However, the GPM view of the users project was very restrictive as was the hierarchical approach to agent communications. The RISKMAN2 system was designed with a narrow view of a software project in mind - that of risk identification and associated advice generation. Therefore the structures of the Risk Taxonomy, GPM, Agent and Blackboard are not readily extendible.

#### **4.4 Desirable Architectural Characteristics**

The architectural approaches discussed above have demonstrated the suitability of an agent-based approach to the support of decision making. There are a number of

interesting lessons to be learned from the above discussion. Specifically the list below comprises the desirable architectural characteristics which should be provided by candidate architectures for an intelligent assistant system for the domain of software project planning.

1. **Specialism** - A multi-agent approach - that of having a community of cooperating agents - provides flexibility in the design of a system. Each agent can be responsible for, or, an expert in, one particular area.
2. **Information exchange** - The multi-agent approach also provides for enhanced information sharing, as agents within the community will exchange information and volunteer information to other interested agents.
3. **Collaboration** - To fully support a multi-agent community, a complex inter-agent communication mechanism must be in place. This represents an overhead in both complexity and system performance.
4. **Blackboard** - The blackboard approach to agent communication reduces the level of complexity and volume of communications traffic.
5. **Supervision** - In an agent community, there does not have to be a single controlling agent. However, the indications are that the existence of such a supervisory agent leads to more organised communications in the community.
6. **Hierarchy** - It seems natural to develop a hierarchy when modelling a multi-agent community. However, the placing of limitations on the operation of agents according to rank within such a hierarchy appears to diminish flexibility, therefore agents should be allowed to operate regardless of rank.
7. **Knowledge evolution** - Using an agent approach to modelling a knowledge base provides the ability to dynamically update the knowledge base by adding, updating and deleting agents. However, the exact internal architecture of the agents is a factor in this flexibility and as such, they must be designed with this in mind.
8. **Data separation** - The separation of data and agents leads to a more open and flexible system. This allows alterations in the data storage mechanisms and the agents themselves to be carried out independently of each other.

9. **Platform independence** - Agents provide an ideal basis on which to develop a platform independent system. ADEPT demonstrated that agents can be successfully implemented in a CORBA distributed environment.
10. **Multiple inference strategies** - If agents in the community are autonomous, they may be implemented in disparate manners and therefore each agent may be implemented in the most suitable fashion for their given purpose.
11. **Component separation** - In separating the agents (which represent the 'intelligence' of the system) from the user-orientated framework (which represents the 'application') the result is a flexible architecture, where either application or intelligent components may be altered independently.
12. **Multiple paradigms** - An agent approach allows the development of a number of different supporting frameworks such as decision support or expert systems. It may also be possible to develop a hybrid framework within which a community of agents could function.

The remainder of this chapter will be devoted to a discussion of the trend towards distributed client-server platform-independent systems and its implications for the development of an architecture for the proposed intelligent assistant system. However, the above characteristics will be further considered in chapter 6 in terms of the proposed architecture.

## 4.5 Architectural Trends

With the increase in globalization, distributed information systems are rapidly becoming the norm. As the need to both compete and cooperate using information systems becomes more clear, system designers are becoming increasingly aware of issues associated with distributed information systems. These systems cross geographical and often organisational boundaries and are typically broadly heterogeneous in both hardware and software terms. Indeed it has been remarked [Orfali et al., 99] that the IT industry stands at a new threshold brought on by; 1) the exponential increase of low-cost bandwidth on Wide Area Networks (WAN), such as the Internet; and 2) a new generation of network-enabled, multithreaded desktop

operating systems, such as Windows NT. This new threshold may mark the beginning of a transition from small (LAN-based) client-server systems, to larger (WAN-based) client-server system that will result in the irrelevance of proximity.

This trend in global distributed information systems and the associated issues of project management is not new. Indeed it has been investigated by projects such as GOAL [Goal, 95], an ESPRIT funded project which addressed the definition of a tool set supporting multi-organisational project management. However, this trend has been intensified by the technological advancements outlined above.

The design and implementation of software is a difficult and expensive activity even where this can be done on a single stable platform using a single operating system and a single programming language. A considerable amount of the software being written now faces additional complexities:

- It must run on a network of machines with the overall functionality distributed among those machines.
- The machines on the system may run different operating systems.
- The components of the system must be integrated easily into new systems, perhaps systems that are not yet planned.
- Legacy systems must be integrated into the system, either to allow the new software to access legacy data or to request the legacy code to carry out some processing.
- It may be necessary to use different programming languages for the components, perhaps because of the use of legacy systems or because a particular programming language is a good choice for some subset of the components.

There are a number of candidate solution technologies for addressing the concerns above, which include; CORBA, DCOM, RMI and sockets. Of these, the two strongest candidates are OMG's CORBA and Microsoft's DCOM, on the basis of both industry popularity and the power of the organisation(s) behind them. However, it is the opinion of many industry observers that CORBA provides a level of openness, flexibility, and industry standardization that brings it ahead of its competitors and as

such provides an ideal basis for addressing the needs outlined above in the context of this research. For an in-depth discussion of these technologies the reader is directed to [Orfali and Harkey, 98], which provides a detailed comparison and critique.

The following sections provide an introduction to the technologies of CORBA and Java. CORBA is investigated as a potential candidate to cater for the needs of distributed client-server systems and Java is introduced as an ideal partner to CORBA to provide the extended flexibility of platform independence.

#### **4.5.1 CORBA**

CORBA [OMG, 96] (Common Object Request Broker Architecture) has two aims; Firstly, it makes it easier to implement new applications that must place components on different hosts on a network or use different programming languages. Secondly, it encourages the writing of open applications, ones that can be used as components of larger systems. The ORB (Object Request Broker) is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

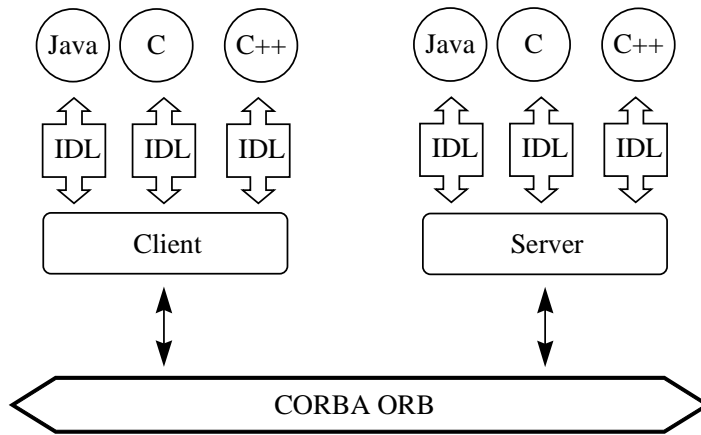


Figure 4.4 - CORBA Interoperability

Within the CORBA framework, all objects are written in a neutral Interface Definition Language (IDL) that defines a components boundaries - that is, the services (or methods) the object provides to potential clients. As IDL is completely language independent, objects specified using IDL are portable across languages, tools, operating systems and networks. IDL is used to specify a components attributes, the parent classes it inherits from, the exceptions it raises, the methods it supports - including the input and output parameters and their datatypes. IDL specified interfaces can be written in any programming language for which there exists a CORBA binding. Essentially IDL allows client and server objects written in different languages to interoperate, as illustrated in figure 4.4.

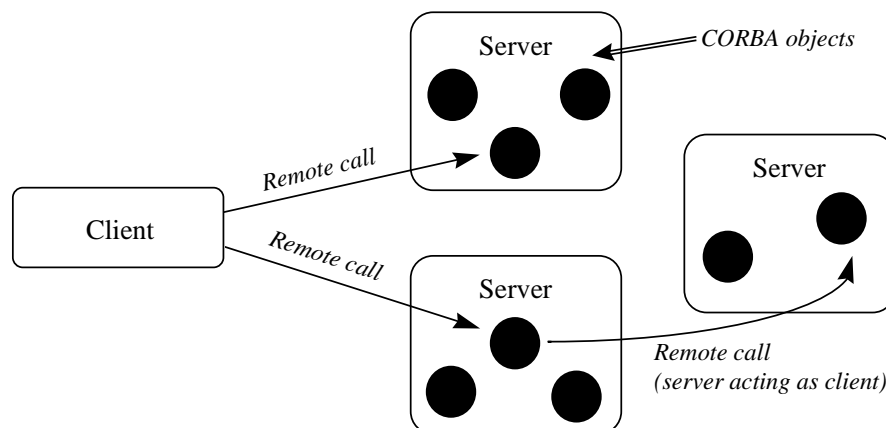


Figure 4.5 - CORBA Client-Server Relationship

Even though CORBA uses the term *client* and *server*, this does not mean that the system must have a star-shaped architecture, where a set of client machines uses a single server. Instead, the objects in one server may use the objects in other servers. This is very useful when decomposing a system into components, because it allows a client to invoke an object in a server, and for that object to invoke on others in order to fulfill the clients request. A server that makes a call to an object remote to it (in another server, on the same or a different machine) is acting as a client for the duration of that call. This relationship is illustrated in figure 4.5

### 4.5.2 The Java Language

The Java programming language was developed by Sun Microsystems in 1990 and since then has created a large amount of hype and discussion in the computer industry. The purpose of the following sections is to show that Java is a platform independent language which is suitable for implementing CORBA based agent applications.

Sun Microsystems define Java as [Gosling and McGilton, 96];

*“A simple, object-orientated, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language”.*

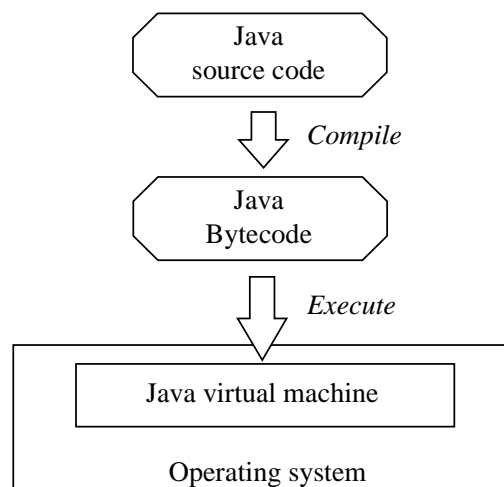


Figure 4.6 - Java Virtual Machine Architecture

Much of the excitement about Java comes from the self-contained, virtual machine environment in which Java applications run. Java compilers generate machine independent bytecode - an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms. The interpreted nature of Java solves both the binary distribution problem and the version problem; the same Java language byte codes will run on any platform. These bytecodes are then interpreted by a Java Virtual Machine (JVM) - a layer of software on a machine, which takes the Java bytecodes and executes them. This results in code that executes the same no matter what its underlying architecture is. Thus Java bytecodes can be shipped over the net and are guaranteed to function the same on all platforms. This relationship is illustrated in figure 4.6.

### **4.5.3 Java as an Agent Language**

Since Java is a mobile code system it is considered by many as an ideal language for the implementation of mobile systems and mobile agents [Srinivas et al., 97]. Mobile code refers to the ability to transfer executable binaries to wherever they are needed to be executed. Though the basic Java support for bytecode migration implies Java code mobility this capability is not completely viable as all Java objects reside on a single host and Java lacks mechanisms for transmitting arguments from one host to another. In contrast, the fundamental premise of CORBA, is that an object on one host can invoke a method of an object on another host, with CORBA employing a referencing model to avoid the issues of object migration.

In distributed systems, the idea of an agent is seen as a natural metaphor, and by some as a development of the concurrent object programming paradigm [Agha et al., 93]. Much of the attention that currently surrounds agent-based technology is related to the phenomenal growth of the Internet. In particular there is great interest in 'mobile agents', that can move around a (local or wide area) network of machines on a users behalf. Many well publicised mobile agents fall into the category of Internet search and filtering agents such as BarginFinder, ShopBot and CyberYenta [Lesnick and Moore, 97].

Recently a large number of languages for agent-orientated systems have been proposed by both the research community and commercial organisations [Muller et al., 99]. Languages such as General Magic's 'Telescript' [White, 94] - which is closely related to the functionality provided by languages such as Java - have failed to gain widespread acceptance. Due to its unique characteristics and overwhelming industrial support, Java is seen by many as providing the ideal candidate for implementing agent-based frameworks and also agents themselves.

#### **4.5.4 Java as a CORBA Object Language**

Java language bindings for IDL provide an application programmer with CORBA's high-level distributed object paradigm. With this in mind, CORBA can be said to bring a number of benefits to Java, as it extends Java with a distributed object infrastructure [Orfali and Harkey, 98]:

- **CORBA provides a scalable server-to-server infrastructure** - Pools of server objects can communicate using the CORBA ORB, and these objects can run on multiple servers, thus providing load-balancing for incoming client requests. The ORB can dispatch the request to the first available object and add more objects as demand increases.
- **CORBA extends Java with a distributed object infrastructure** - CORBA allows Java to communicate with other objects written in different languages across address spaces and networks. In addition, CORBA provides a rich set of distributed object services that augment Java - including transactions, security, trader and persistence.

The Java infrastructure starts where CORBA ends. CORBA provides a distributed object infrastructure that lets applications extend their reach across networks, languages and operating systems. Java provides a portable object infrastructure that works on every major operating system. CORBA deals with network transparency, while Java deals with implementation transparency. Java offers a number of attractions for implementing CORBA objects:

- **Java allows CORBA to move code around** - Java's mobile code facility allows code to dynamically move across the CORBA infrastructure to where it is needed, thus allowing clients and servers to dynamically gain behavior.
- **Java simplifies code distribution in CORBA systems** - Java code can be deployed and managed centrally from the server. This means that code on the server is updated once and clients can receive it when they need it.
- **Java complements CORBA's agenting infrastructure** - CORBA defines a framework for distributed objects which lets 'roaming objects' move from node to node. Java bytecodes are ideal for shipping such behavior around.

Orfali [Orfali and Harkey, 98] states that "*Java is almost certainly the ideal language for writing both client and server objects*". Java's built-in multi-threading, garbage collection and error management makes it easier to write robust network objects. Also, Java's object model complements CORBA's, as they both use the concept of interfaces to separate an objects definition from its implementation.

## 4.6 Summary

This chapter has presented the architectural aspects of intelligent systems and discussed the trend towards distributed client-server platform independent systems. A number of intelligent assistant systems were investigated and the concept of agent-based architectures for intelligent assistance presented. In addition, CORBA and Java were presented as a solution technology to the issues previously identified.

Chapter 5 will outline the issues surrounding the knowledge base which forms the basis of the intelligent agents for the proposed system. There will also be an examination of the selection of a suitable method for acquiring and representing knowledge.