Distributed Software Systems: From Objects to Agents

Martin Purvis, Stephen Cranefield and Roy Ward
Department of Information Science
University of Otago, Dunedin, New Zealand
Email: {mpurvis,scranefield,rward}@commerce.otago.ac.nz

Abstract

With the rapid increase in both the amount of available information and extent of computer interconnections, there is a growing interest in systems that can provide more efficient and flexible access. This paper describes research and development into open distributed software systems frameworks designed to (a) provide access to legacy systems and (b) accommodate the introduction of new systems without requiring recompilation. The focus of this research is to use existing distributed object technology and extend it with ideas from agent-based software engineering. An example of this work, as it is applied to the development of an open environment for carrying out connectionist computations (such as neural network analysis), is discussed in detail.

1. Introduction

The past few years have seen an acceleration in the amount and availability of electronically stored information. This information explosion is due to several factors: greater use of automated data acquisition techniques, significant cost reductions in data storage technologies, and telecommunications developments that have enhanced the inter-connectivity and efficiency of data transfer among distributed sites. The increased inter-connectivity also means that there is now a potentially widely-distributed array of processing modules available that can be applied to various computational tasks. However, although the expanding information repositories are increasingly available to wider circles of society, they are stored by means of various media types (text, audio, images and other digital media, *etc.*), differing formats (flat files, relational databases, object-oriented databases, *etc.*) and organised according to differing semantics. In addition the computational modules that are increasingly available across the network each have their own interfaces and protocols that must be known by clients for effective interaction.

The overall result of these developments has been that the expansive growth in the volume of available information and processing modules has outstripped the capacity to organise, access, process, and efficiently interpret the required information. Our research goals are concerned with the problem of how to organise larger information systems in the context of the open, expanding distributed environment in which we find ourselves. In particular it is our intention to

- take advantage of existing technological developments that are becoming available, and
- at the same time, be able to access legacy systems that may have been developed according to the assumptions of an earlier technological era.

Our proposed solution is to develop a computational infrastructure that is schematically depicted in Figure 1. Various sources of information are shown at the bottom of

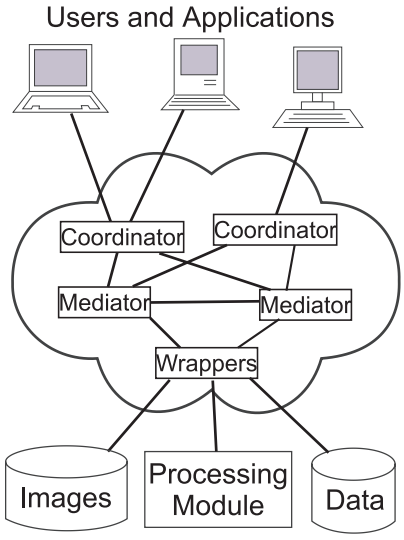

Figure 1. Integrating distributed information

the figure and may be in the form of flat files, maps and images, sound, etc. In order to integrate this information and raise it to a level of abstraction suitable for practical use, several types of operations are performed. First individual drivers or wrappers are provided that make the information available to mediator modules. The mediator modules use encoded knowledge about the semantic content of the data to create information suitable for higher-level applications. The coordinator module combines information from individual mediator modules and makes the information available to applications.

Some of this computational infrastructure may be developed with the use of existing commercially available object-oriented technology. In this paper we discuss research work at the University of Otago in association with the Connectionist-Based Information Systems (CBIS) and Distributed Information Systems (DIS) research projects that involves the use of the Common Object Request Broker Architecture (CORBA) to implement elements of the scheme depicted in Figure 1. A discussion of this work is provided in Section 2, which covers an overview of CORBA, and Section 3, which describes the distributed system that has been developed at Otago.

In order to go further towards distributed open-systems, however, it is necessary to develop information protocols that can accommodate a more general exchange of information concerning the distributed data sets and processing modules. This information may be meta-data, that describes various characteristics of the data, or it may be other types of information that characterise the nature of any interactions that can be performed with a given processing module. In order to move to this next level of distributed information systems processing, it is appropriate to employ an agent-based architecture. In this case the coordinator, mediators, and wrapper-encapsulated tools or datasets are implemented by means of software agents which negotiate among themselves in order to access information and processing modules appropriately. Section 4 provides a discussion of agent-based software engineering notions. Section 5 discusses how the Otago Connectionist-Based Information System software is being extended with elements based on these ideas.

2. CORBA

Object-oriented technology is recognised as providing to computer information system developers the advantages of modularity, data-hiding, application-level abstraction, and reuse. Until recently, though, object-oriented systems have largely been developed on a single platform and applied to a single set of tasks. CORBA [1] is an effort on the part of a consortium of computing enterprises to create a standard for object-oriented operation in a distributed environment. CORBA provides a software 'bus', which is implemented by means of ORBs (Object Request Brokers) shown in Figure 2, that enables a client application to access another object by name anywhere on the network, without having to know its location or being aware of the mechanisms used to communicate with or access server objects. In order to access the ORBs appropriately, it is only necessary to write stubs and skeletons, using the CORBA Interface Definition Language (IDL), to make the connections operational.

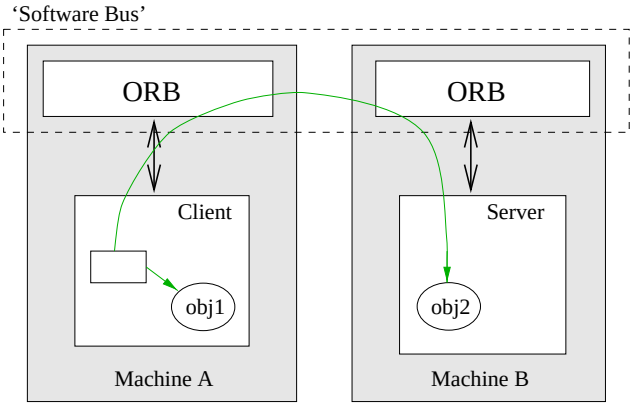

Figure 2. CORBA calls

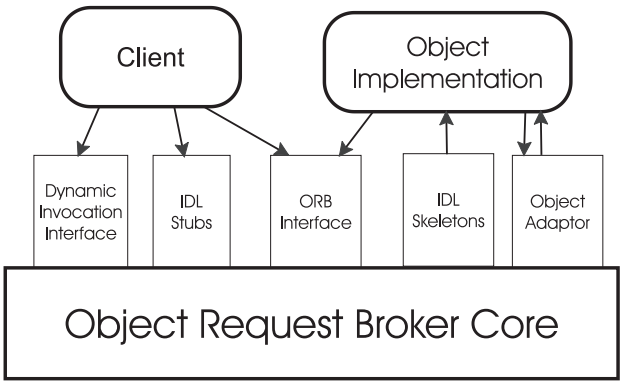

Figure 3. The CORBA architecture

The CORBA infrastructure for distributed systems is illustrated in Figure 3. CORBA objects can be located anywhere on the network, and the language, compiler, and operating system used to create them are transparent to the clients that use them. To achieve this transparency it is first necessary to provide wrappers around legacy code that is written, say, in COBOL or C++ and make it callable by means of object method invocations. This is accomplished by using IDL-specified interface code and enables the developer who is invoking these objects to use native language constructs for his or her method invocations. The

Dynamic Invocation Interface shown in Figure 3 enables clients to call method operations on objects without knowledge at compile-time of their IDL type, *i.e.* without having the IDL stubs available. Thus it is possible with the DII to discover methods to be invoked at run-time. The Object Adaptor provides a standard component for activating objects (on the server side) that are required by client method calls. The ORB Interface shown in Figure 3 provides a standard set of services that can be of use to CORBA clients and servers, such as converting object references back and forth to strings. CORBA thus provides an industry standard for implementing some of the elements shown in Figure1: for example, the wrapper interfaces can be implemented using IDL.

3. Distributed objects at Otago: CBIS

3.1. Motivation

The CBIS (Connectionist based Information System) is designed to be a comprehensive set of tools and methodologies for performing data analysis and research into data analysis, mostly using artificial neural networks and other approximate techniques [2]. Since it is a collaborative project involving several groups working in universities and industries around New Zealand, it is important that developed software be

- *Distributed*, so these groups can contribute parts of it, and anyone can have access
- *Modular*, so that a large range of tools can easily be added by different people
- *Multi-platform*, to maximise the options available to developers, and take advantage of the most appropriate or available hardware
- *Scalable*, in that many of the data sets used for training neural networks are extremely large, so it may be impractical to have an entire data set located on a machine performing a neural network computation
- *Easy to use*, in that the distributed nature of the system should be transparent to the user

Ideally, an analyst would prefer to have a system that could flexibly incorporate new connectionist modules located anywhere on the network, without having to recompile or reconfigure his or her modelling environment. Additionally, because of the computationally expensive (and therefore often lengthy) nature of some neural network computations, it is desirable to support coarse-grained parallelism.

3.2. How the CBIS is implemented

The CBIS architecture is a distributed framework consisting of a collection of modules, each module performing a specific computational task such as neural network computation, fuzzy logic inference or data manipulation.

The modules in the CBIS Architecture share common protocols and are connected together over a network using CORBA. Each module is a CORBA object, and communication is set up as client-server relationships: a module calls a method of another module to perform a service, and a result is returned to the calling module.

The tasks that the modules are required to perform can be broken up into the following categories:

- *Computation*, to operate on data. This may be a *wrapper* around some legacy code as described in Section 1
- *Data*, to store or stream data ready for use
- *Data Access*, to store and retrieve data in external repositories. This is another example of a *wrapper*
- *Control*, to manage the other modules. This is an example of a *coordinator*
- *User Interface*, to allow tasks to be given to the system to perform.

A diagram of the overall CBIS architecture is given in Figure 4. All interactions of the system that take place across modules connected by thick arrows in Figure 4 involve communication by means of CORBA ORBs. The User Interface enables the user to drag icons representing data objects into drop boxes representing computational modules without having to know the locations of the objects associated with these icons. Data objects and computational modules are CORBA objects that are accessed through CORBA calls and may exist on any host on the Internet that is running a CORBA ORB. Note that this architecture currently has a coordinator (the control module) and wrappers (data access and some computational modules), but no mediators.

An existing connectionist computational module developed elsewhere can be used by the system by wrapping it with appropriate CORBA interface code as described in Section 3.3. As a result, there may be more than one module available to accomplish a particular task, for example there could be many backpropagation modules existing on different machines, any of which could be used for this computation.

When a task is to be performed, the data required for the task, the tool to perform it and any required parameters must be on the same host. Consequently the ability to move data objects around as required is one of the key features of the architecture.

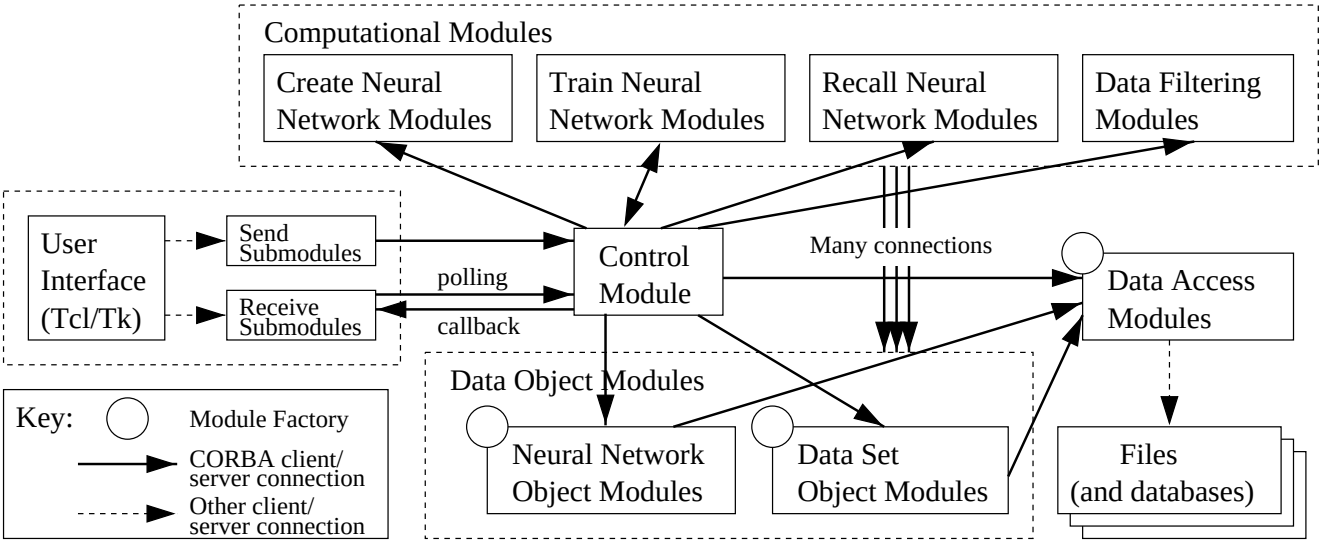

Figure 4. The architectural components

One of the features of the system is that there are many sorts of parameter and status information that are passed around, for example performing training by backpropagation on a neural network requires a learning rate and other parameters. While it would be possible to have a different interface for each module so that it is passed the specific information it needs, this is clumsy and would limit the extensibility of the system, as the control module would need to know about each type of parameter block. Instead, this system opts to use a message structure inspired by the Knowledge Manipulation and Query Language (see Section 4.1). An example of such a message for network training might be:

```
(achieve :reply-with test
   :content
      (train :method (backprop
                        :learning-rate 0.5
                        :momentum 0.5
                        :max-epochs 500)
             :data-set 1
             :input-network 2))
```

3.3. Adding new tools to CBIS

It is important to be able to use tools designed for other systems so as to get maximum code reuse. This can be done by wrapping a tool in a piece of code that handles the CBIS protocols. For example, using code in MATLAB [3] that performs a particular task, a CBIS module can be constructed as in Figure 5 that behaves just like any other CBIS module. The module is passed the location of resources and a message as described in Section 3.2, the wrapper converts these into a form suitable for the wrapped tool, then converts the tool's results back into a form which CBIS can use.

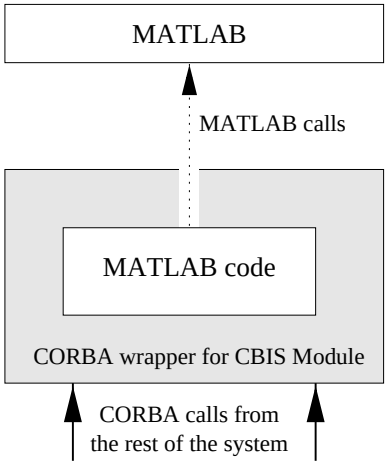

Figure 5. A CBIS wrapper

4. From objects to agents

In order to provide more general distributed system services it is appropriate to employ agent-based software engineering techniques. This section discusses the field of agent-based software interoperation and how this work can be used to enhance the CBIS system.

4.1. What is an agent?

The term ‘agent’ is becoming increasingly prevalent in the computing literature, descriptions of software products and even the popular press. Because of its current buzzword status and the obvious differences between the various types of system being proclaimed as agents, a number of authors have attempted to provide a taxonomy of agent systems or to identify the properties and abilities they feel characterise an agent [4–7]. Possible criteria range from the dictionary definition of an agent as “one who acts for another in business, *etc.*” [8] to combinations of more specific requirements such as autonomy, reactivity, proactivity, mobility and/or having a knowledge-level representation of the world state and some goals to be achieved.

Some common themes in ‘agent’ research are the development of mobile agents (where the principal, and sometimes only, requirement is that the system can relocate itself to another machine as part of its life cycle), user interface agents (where the agent acts as a proxy for a user by automating his or her interactions with one or more software systems), multi-agent systems (where the research focus is on theories and mechanisms for inter-agent communication, cooperation and negotiation) and software engineering agents (where the emphasis is on simplifying the analysis or construction of interacting distributed systems based on a communicating service-providing agent as a structuring abstraction). The following discussion uses the term ‘agent’ in the latter sense, in particular according to its use in the field of agent-based software interoperation [9].

Agent-based software interoperation (ABSI) is based on the idea that “communication can best be modelled as the exchange of declarative statements (definitions, assumptions and the like)” [9]. In order to facilitate the coordinated use of disparate software tools that were not designed to work together, the ABSI paradigm involves equipping tools with wrappers that allow them to communicate in a common knowledge-level communication language instead of in terms of low-level vendor-specific data files or streams. These communications, expressed using the Knowledge Query and Manipulation Language (KQML) [10], are messages specifying a *performative* (a concept from speech act theory denoting the intent of the message, *e.g.* to inform or request [11]), and containing arguments representing the information content of the message and additional details such as the sender and intended recipient of the message, the knowledge representation language used to express the message content, and the *ontology* that should be used to interpret the terms appearing in the content expression — a formal and publically-available specification of the terminology used to describe a given domain.

KQML is designed to be a standard[1] language for inter-agent communication, and together with the use of a standard content language such as Knowledge Interchange Format (KIF) [13] and published ontologies defining entity names, functions and relations for common domains, this provides the semantics for inter-agent messages. Thus, agent designers do not need to study technical documents describing the public interfaces of other peoples’ agents in order to ensure that their own software can communicate effectively with them. Provided that two agents share a common ontology (and are implemented so that their responses correctly describe their computations in terms of that ontology) they can use each other’s services[2].

In the field of ABSI an entity is considered to be an agent if it can communicate with other agents using KQML (there are no requirements on its internal architecture which may simply be an existing software tool encapsulated within a new KQML-speaking wrapper). However, to allow agent systems to be extensible it must be possible for agents to locate each other by name (rather than address) or by the services they offer. KQML therefore defines performatives to allow agents to *advertise* their services to a special *facilitator* (or matchmaker) agent which can then handle requests from agents looking for these services, either by *recommending* the service agent to the requester, *recruiting* it to perform that service or *brokering* the service on behalf of the requester [14]. Figure 6 shows the architecture of an ABSI system. In the original proposal for ABSI systems [9] there was a facilitator on each host and all communications were routed via these facilitators. However, some implementations of ABSI ideas allow agents to communicate directly once they have each other’s addresses.

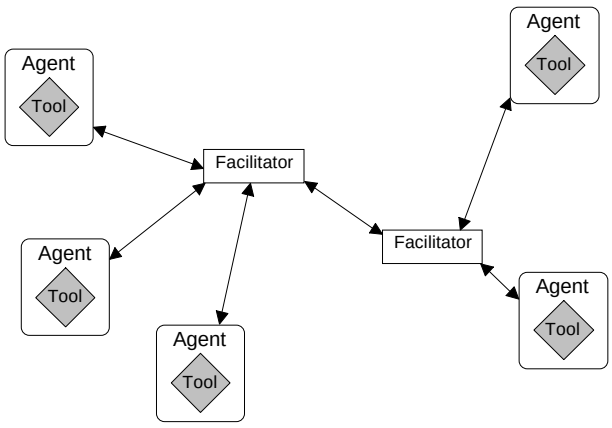

Figure 6. ABSI architecture

1. Although KQML has become a *de facto* standard, the Foundation for Intelligent Physical Agents has recently proposed an alternative, but similar, agent communication language [12].

2. The problem of translating between different ontologies for the same domain is also an active research area.

Table 1. CORBA and ABSI system features

Feature	CORBA	ABSI Systems
Status	commercial implementations available	various research architectures
Message type	method call	performative + content
Message semantics	procedural	declarative
Remote name-based message routing	yes	yes
Service-based matchmaking	proposed trader service	facilitators
Extensibility	components must know each other's external interfaces	components must speak KQML and some advertised ontology

4.2. Distributed objects *vs.* agents

A common question that arises in the context of ABSI is "what is the difference between distributed object systems, such as CORBA, and ABSI agent systems?" Both provide infrastructure allowing the encapsulation and interoperation of disparate and distributed software systems. In fact, a distributed object architecture such as CORBA already provides a convenient transport layer on which an ABSI system could be built, and future versions are likely to offer agent-level services such as service-based client-server matchmaking, mobility, etc. Our view is that an agent is a specialised type of object that communicates in terms of high-level declarative messages. The maximum flexibility and extensibility of a distributed system is possible if all components forgo any ability to make unconstrained method calls to each other and restrict themselves to a standard high-level communication language. This allows the use of matchmaking and other potential coordination services such as goal-based planning for the sequencing of agent invocations (see Section 5.2). This will also allow the system to make use of specific technologies and theories developed by the research community to support inter-agent communication, cooperation and negotiation. Table 1 compares some of the features of CORBA and prototypical ABSI systems.

5. Adding agent services to CBIS

Compared with the general services provided by agent-based systems, the current CBIS system has some limitations. The control module has to know the location of all of the current modules and resources, and initiate all communications between modules. Modules have no autonomy. Currently information to handle this is coded into the control module. It is possible to use a name server to store the location of modules, but that still leaves the control module to handle the interactions between other modules, making the control module very complex.

In some circumstances tasks may be easy to factor into local components, or some Internet locations may have special requirements such as security. In these cases, it may be useful to have a distributed control structure involving several control modules communicating with each other. In the current system this is not possible, since there has been no provision made for multiple control modules to interact.

Some form of facilitator as described in Section 4.1 can be used to provide the ability to modularise and distribute the functions of the control module.

Another limitation is that each type of CBIS module (training, recall, data transformation) has a specific method call required to use modules of that type. This poses difficulties if there is to be a new category of module added, say one that requires three data sets to produce one new one, or uses a new type of object. Currently, the control module would have to be altered to handle this new case. To overcome this problem, it would be possible to incorporate all of the parameters into a single structure (for example the KQML-like structure of Section 3.2) to give all of the method calls the same structure. Another solution is to use the CORBA Dynamic Invocation Interface, but this is somewhat unwieldy.

There are solutions to these problems that can be coded individually or may be provided by CORBA (or future versions of CORBA), but a framework for dealing with them in a systematic way is preferable. ABSI agents using KQML with a well-defined ontology provide such a framework.

5.1. Using CBIS from other systems

In order to make the best use of the CBIS, it is essential that CBIS modules be available as general tools able to be used by other systems rather than being tied in to one specific control architecture. For example, consider a data processing system *S* that doesn't have a 'shuffle data' function (neural network training data is often shuffled before presentation). The user wishes to have system *S* use the CBIS to shuffle the data, where the CBIS takes care of all of the data conversions as shown in Figure 7. Using CBIS modules directly would be unworkable, as the other system will have no way of locating CBIS modules, and even if it did the code to do even a simple task is highly complex without the support provided by the control module. Using the CBIS control module would be better, but still has the problem of not being designed for the job — it is designed for a CBIS user interface to be attached which makes it awkward for this sort of problem.

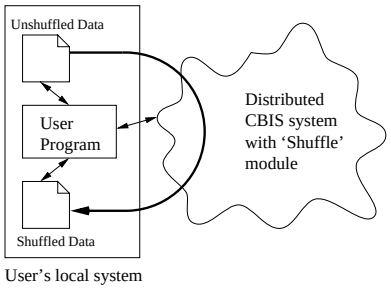

Figure 7. Using CBIS to shuffle a data file

The solution to this is to have a facilitator module that receives a request from *S* to perform a task and then takes all of the steps required to do that one task. This includes retrieving the required information, processing it, returning the results to *S* and handling any data conversions on the way. The facilitator is an ABSI agent using KQML as described in Section 4.1. Note that this agent makes use of the CBIS modules as they stand, so the CBIS modules don't need to be wrapped to form agents — the facilitator handles external communication.

In the example of shuffling a data file, the following takes place as shown in Figure 8. System *S* first sets up servers that the CBIS can use to read an input data file and write an output file. These are simple CORBA objects that transmit or receive the file as plain text and transfer it from or to the file. *S* then locates a CBIS facilitator, and sends a message containing the following information:

- the CORBA reference of the server on which the unshuffled data resides
- the file format of the unshuffled data (called *F*, say)
- the CORBA reference of the server to which the shuffled data should be sent
- the file format of the shuffled data (called *G*, say)
- the operation to be performed

The message has the following form, where *RefInput* and *RefOutput* are strings refering to CORBA objects, pKQML is a content language similar to KQML, *SpecInformation* is some information about the data such as the size, and *F* and *G* are file formats:

```
(evaluate
     :language pKQML
     :ontology data-manipulation
     :reply-with done
     :content (shuffle
          :input (RefInput :format F)
          :output (RefOutput :format G)
          :specification SpecInformation)
     :sender S)
```

The facilitator creates some data objects as storage for the data, finds CBIS modules for data format conversion (both ways) and a module to perform the shuffle as shown in Figure 8. It then initiates transactions to move the data along the path shown by the thick arrow, which are done independently of the user program.

S, having waited for the facilitator to signal that it is finished, now shuts down the servers that it set up for the data transfer.

This design shows how a client-server system based on distributed objects can be made more flexible by extended it to include agent services, and how this can use existing client-server modules.

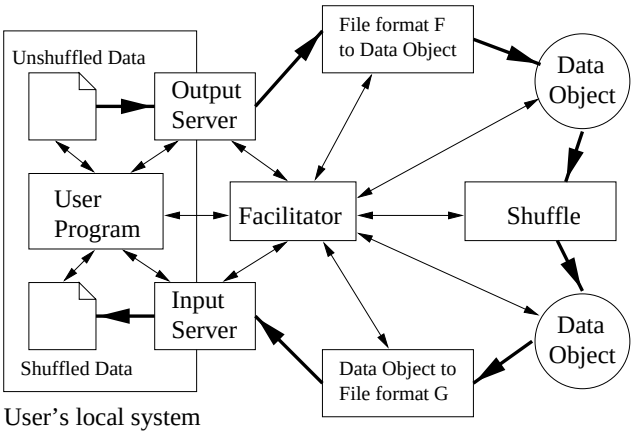

Figure 8. Details of shuffling a data file

5.2. Automating patterns of tool interoperation

The Connectionist-Based Information System provides a framework for integrating a collection of disparate data analysis tools. Problem-solving with such a toolkit often involves an exploratory approach where alternative tools may be used in order to find out which techniques work best for the current problem. However, although the complete sequence of tool application necessary to solve the problem may depend on the skill and intuition of the user, there are likely to be common patterns of tool use. Also, data format transformations and data transfer from one host to another may be required in order for the results from one tool to be input to another. In order to relieve the user of the overheads in time and memory to remember the correct sequence of actions and then manually invoke these tools and data transformations, another project at Otago is investigating the extension of the ABSI architecture by adding a user agent (to record the user's current goals and control the execution of agent actions) and a planning agent (to generate sequences of agent actions based on the user's current goal) [15–17]. It is intended to apply these ideas to the CBIS project so

that for different types of data analysis (*e.g.* static or time series classification, time series prediction, *etc.*) the user can trigger alternative exploratory strategies which are then automatically planned and executed.

6. Distributed Objects at Otago: DIS

The Distributed Information Systems project is a recently initiated collaborative research programme that involves the University of Otago, New Zealand Crown Research Institutes and the Distributed Systems Technology Centre of Australia. Its goal is to research and develop prototype systems that can connect various disparate and distributed databases in New Zealand. For this work a facilitator agent will serve as a service registry and brokering facility for other agents, such as a coordination agent, an ontology agent, a data analysis agent, a task planning agent, *etc*. Like the CBIS project system, the DIS system will be implemented by using CORBA to interconnect software modules, which will primarily be existing user applications and data repositories around New Zealand. Here, though, rather than to provide an open environment for connectionist computations (as with CBIS), the goal is to provide an even more general framework for the interconnection of many sorts of existing applications and data sets. As such, this work will necessarily involve more extended use of and research into agent-based software engineering techniques.

7. Acknowledgements

This work has been funded by the New Zealand Public Good Science Fund and the Otago University Research Committee.

References

[1] J. Seigel. *CORBA Fundamentals and Programming*. John Wiley and Sons, Inc., 1996.

[2] R. Ward, M. Purvis, R. Raykov, F. Zhang, and M. Watts. An architecture for distributed connectionist computation. In *Progress in Connectionist-Based Information Systems*. Springer, 1997.

[3] The MathWorks, Inc. *Using MATLAB*, 1996. http://www.mathworks.com.

[4] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10, 1995.

[5] L. N. Foner. What's an agent anyway? - a sociological case study. Report available by FTP, MIT Media Lab, May 1993. file://media-lab.media.mit.edu/pub/Foner/Papers/What's-an-Agent-Anyway–Julia.ps.Z.

[6] H. S. Nwana. Software agents: An overview. *The Knowledge Engineering Review*, 11(3), 1996.

[7] C. J. Petrie. Agent-based engineering, the Web, and intelligence. *IEEE Expert*, 11(6), December 1996.

[8] R. E. Allen. *The Pocket Oxford Dictionary*. Clarendon Press, Oxford, 7th edition, 1984.

[9] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.

[10] T. Finin, R. Fritzson, D. Mckay, and R. McEntire. KQML: An information and knowledge exchange protocol. In Kazuhiro Fuchi and Toshio Yokoi, editors, *Knowledge Building and Knowledge Sharing*. Ohmsha and IOS Press, 1994.

[11] J. R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, 1969.

[12] Foundation for Intelligent Physical Agents homepage, September 1997. http://drogo.cselt.stet.it/fipa/.

[13] Knowledge Interchange Format specification. Working Draft, ANSI X3T2 Ad Hoc Group on KIF, March 1995. http://logic.stanford.edu/kif/specification.html.

[14] D. Kuokka and L. Harada. Matchmaking for information agents. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, volume 1, pages 672–678, 1995.

[15] S. J. S. Cranefield and M. K. Purvis. Agent-based integration of general-purpose tools. In *Proceedings of the Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, December 1995.

[16] S. J. S. Cranefield and M. K. Purvis. An agent-based architecture for software tool coordination. In L. Cavedon, A.S. Rao, and W. Wobcke, editors, *Intelligent Agent Systems: Theoretical and Practical Issues*, number 1209 in Lecture Notes in Artificial Intelligence, pages 44–58. Springer, 1997.

[17] A. C. Díaz, S.J.S. Cranefield, and M.K Purvis. Planning and matchmaking in a multi-agent system for software integration. *Mathematical Modelling and Scientific Computing*, 8, 1997.