UNISYS

How to Serve Knowledge

Notes on the Design of a Knowledge Base Server

Tim Finin
Rich Fritzson

PRC-LBS-8906
March 1989

Paoli Research Center
P.O. Box 517
Paoli, PA 19301

How to Serve Knowledge

Notes on the Design of a Knowledge Base Server *

Tim Finin
Unisys Paoli Research Center
P.O. Box 517
Paoli, PA 19301
215-648-7446
finin@prc.unisys.com

Richard Fritzson
Unisys Paoli Research Center
P.O. Box 517
Paoli, PA 19301
215-648-1328
fritzson@prc.unisys.com

December 1988
revised February 1989

Abstract

This note describes work on the design of a *Knowledge Base Server* - a distributed architecture for delivering knowledge representation and reasoning services to applications. The idea is simple: knowledge bases are like databases and their services should be provided in a similar manner – in a client-server relationship. This paper motivates the idea, discusses some of the design issues, and briefly describes our current approach.

1 Introduction

Knowledge bases are best viewed as being akin to databases; they are global resources to be shared by users and application programs. A knowledge representation and reasoning (*KR&R*) system should offer its services to programs in the same way that a database does: as an independent process exchanging information via network streams or interprocess communication pipes. Figure 1 shows a knowledge base server attached to a local area network.

The two traditional approaches to providing *KR&R* service to an application require either that the application be written in the language and environment of the AI system, or that the AI system be provided in the language and environment of the application. Often, neither of these alternatives is acceptable because the application can not be moved from its environment and because the AI systems available in that environment are inadequate.

A knowledge base server provides a third alternative for embedding AI in an application. When a knowledge representation and reasoning system is structured as an independent server process, it offers several advantages:

- **Applications can be smaller.** The application no longer needs to contain the general *KR&R* routines, but only the application-specific representation and inference procedures.
- **Access to the knowledge base is language independent.** There is no need to program in, or even be aware of the language in which the knowledge base is implemented. This allows both the application and the *KR&R* system to be implemented in the most appropriate language.
- **Persistent and sharable knowledge bases are easier to manage.** Having a single server for multiple users makes dealing with concurrency and locking easier.
- **The knowledge server can offer a uniform front-end to conventional database management systems.** If the knowledge representation system is appropriately designed, it can also serve as a front end for database management systems that are attached to the network. (See Figure 2.)

Knowledge Server Goals

The primary goal for a knowledge server is to provide full support for an application's *KR&R* requirements, even though the knowledge server is executing on a separate machine. If we meet this goal, we can expect to reap the advantages described above. However, a good knowledge server will be designed to address several other goals as well. In particular,

- A good knowledge server will be designed to reduce the overhead of the communications links as much as

*A version of this paper appeared in the AAAI 1989 Spring Symposium on Knowledge System Development Tools and Languages, Palo Alto, March 1989.

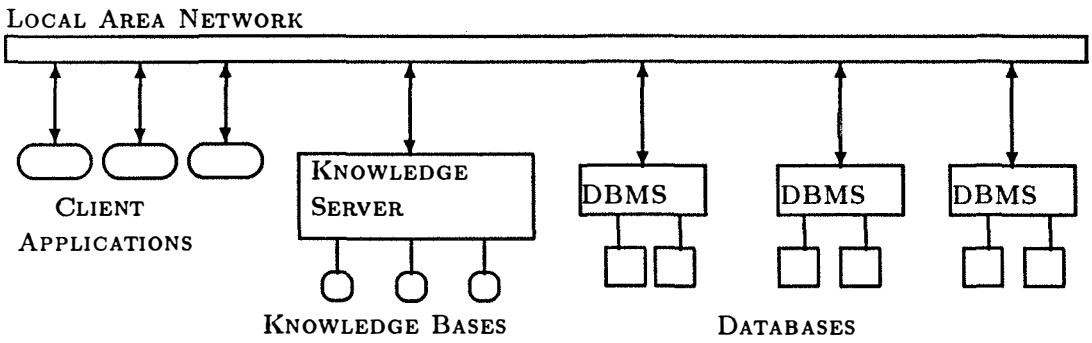

Figure 1: Physical Design of Server Architecture

possible, and will encourage taking advantage of the parallelism inherent in a cooperating process design.

While communication links are generally quite fast, a good design will attempt to reduce traffic on it by:

- eliminating redundant transfer of information,
- using a compact encoding of information,
- allowing the user to augment the knowledge server with custom reasoning routines *which execute on the server's host.*

Allowing the user to augment the reasoning processes of the knowledge server will make the system significantly more useful as well as allowing for efficiency improvements.

- The knowledge server should be extensible.

It should allow the user to extend the reasoning procedures either by writing code for the knowledge server (to be executed on the knowledge server's host), or by writing application code (to be executed on the application's host) which can supplement the reasoning procedures of the knowledge server.

Knowledge Server Size

In order to produce a flexible and useful knowledge server, there are several goals which should be kept in mind. The first is that we should be trying to provide a *big* system which supports a rich variety of knowledge representation structures and inferential procedures. The second goal is to provide a system which can also be *small*, that is, one which can be configured so that only the modules which are needed by an application are actually loaded, making delivery on modest machines possible.

There are two senses to the notion of size in knowledge-based systems — the size of the knowledge bases that can be effectively managed and and the size of basic *KR&R* system itself. In fact, it is reasonable to consider these to be independent dimensions along which a particular KB system (or a requirement for one) can vary.

Figure 3 shows the two-dimensional space defined by these two notions of size and places some sample points on them. The Kandor system [8,7] is an example of a KR system that was explicitly designed to offer a small, well defined set of representation capabilities with a simple, functional interface. The Kandor system did not, however, address any of the issues which arise in managing very large knowledge bases. Postgress [9] is an example of a system which attempts to offer efficient access to very large knowledge bases by extending and enriching the traditional database technologies to include some support for knowledge-based operations such as inference rules. These extensions result in an extremely simple and rudimentary knowledge representation system, however.

The KEE system [5] is a good example of a very large, rich *KR&R* system which has no special considerations for supporting very large knowledge bases. The very fact that the representation and reasoning system is large and complex makes systems like KEE poor performers on very large knowledge-bases. The general correlation between the simplicity of the knowledge representation system and the size of the knowledge-base that it can support is reflected by our placement of the NIKL [4] system in this space. NIKL lies somewhere between Kandor and KEE with respect to the complexity of the representational constructs and inferential services it offers. It also is positioned between Kandor and KEE with respect to the size of the knowledge bases that it can practically support. The last sample point in the figure is CYC [6] which is somewhat unique in its goals. CYC employs a very complex and large knowledge representation system but is also explicitly targeted toward supporting extremely large knowledge bases. It's success at combining both aspects

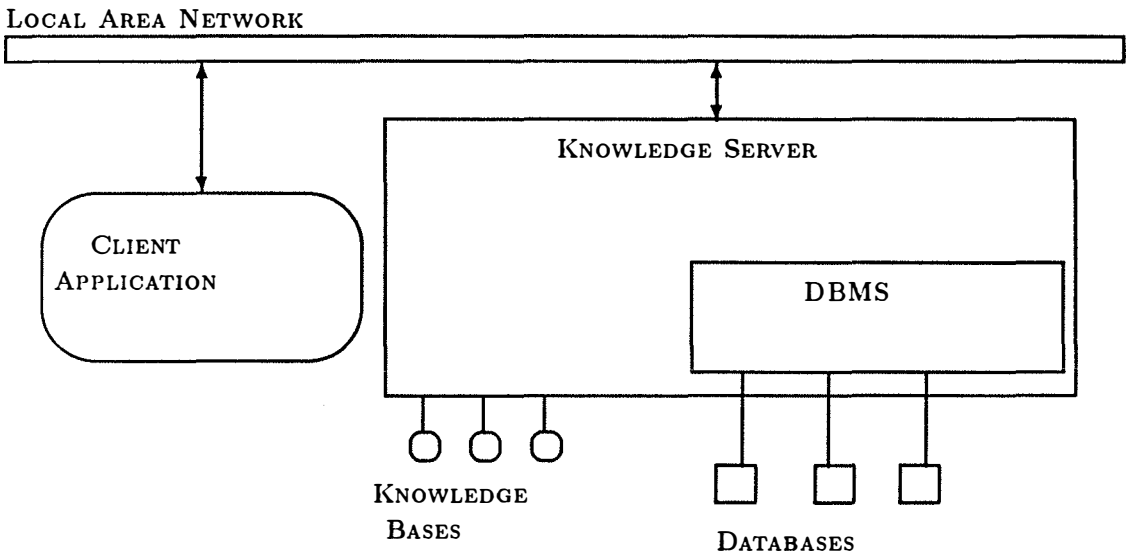

Figure 2: Logical Design of Server Architecture

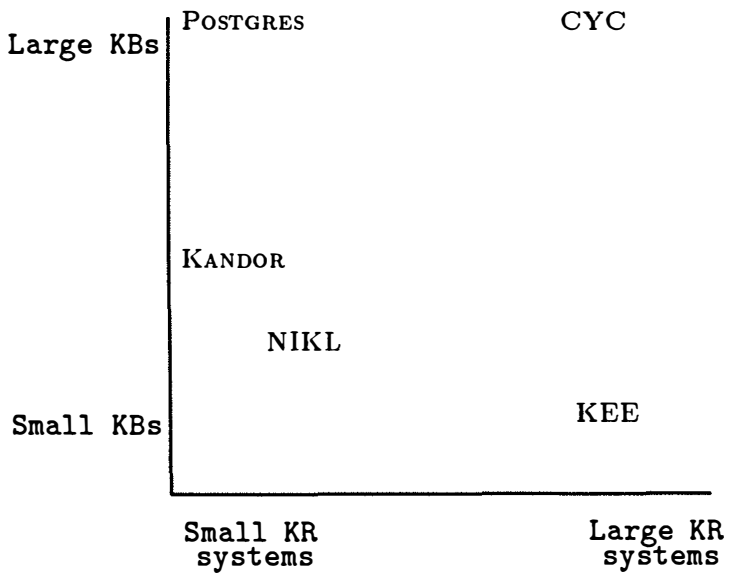

Figure 3: Size of a KBMS system

of bigness has not yet been tested.

Our goal is to provide a KBS system that can be both *large* and *small* in both of these dimensions. We would like to provide a system that can be *large* in that it supports a full range of standard knowledge representation structures and procedures which can operate efficiently on knowledge bases of significant size. Both frame-based and rule-based representations, with forward and backward chaining inference mechanisms, and some form of truth maintenance, should all be available to application programs. In addition there should be modules that supply other functions on knowledge bases, such as an editor, a knowledge base browser, classifier, consistency verifier, etc. The system should also be large in the sense of being able to manage large knowledge bases, including knowledge bases which are stored, in part, on external databases. Running on a fast "server system" host, it should be capable of supporting the development of large AI projects.

The second goal is to provide a system which can also be *small*, that is, one which can be delivered. It should be possible to configure the system so that only the modules which are needed by an application are actually loaded. It should also be possible to execute the knowledge base server on the same host as the application program. In such a configuration, the penalty for the server architecture should be minimized.[1]

Feasibility

There are several objections that can be raised against the knowledge server architecture.

- It will be too slow. Transferring the data over the network will reduce performance to unacceptable levels.

This is not likely to be true. Since the reasoning component of the system is on the same side as the knowledge, the amount of information transferred between the application and the server should not be that great. In fact, less information should pass between the knowledge server and the application than would usually pass between a conventional application and a database. These conventional applications do not find the network connection to be a bottleneck.

For applications which do need to transfer large amounts of data between a client application and the knowledge server, the data transfer rates of current technology networks is approaching the transfer rates of large hard disks. Even information intensive applications such as the X Window System can operate acceptably over a network. So the speed of the network link is not likely to be a bottleneck for the application.

In fact, a well designed application could actually run faster with a knowledge server because of the opportunity for parallelism that the architecture introduces.

- Applications are too tightly coupled with the knowledge representation and reasoning components to be divided by a low bandwidth connection.

This objection is more serious. However, the knowledge server design doesn't try to separate general purpose reasoning from knowledge representation; that would be too difficult and would be likely to fail. On the other hand, most real applications utilize a mixture of general purpose reasoning and special purpose reasoning. The latter can be simple or complex; it may only be called once or it might have to be available as a subroutine to be called by the general purpose reasoner, or it might have to be available as a coroutine, maintaining an internal state and swapping control back and forth between the knowledge server and itself. There are several ways to deal with this requirement, some of which we discuss in the next section.

2 Design Issues

A functional interface to a knowledge representation system has to provide access to the services provided by the system, such as its frame representation or its backward chaining reasoning mechanism. But when the interface is going to be provided via an interprocess stream to an independent process, rather than via procedure calls to code within the same process, new issues arise. Among these are:

- Can the server process handle more than one user simultaneously?
- Can the server process handle more than one knowledge base?
- What communication protocol will be used between client and server?
- How will data be passed from the server to the client's programming language?
- How can the client supplement the server's inference processes? Will it be possible to use the client's programming language to do this?

Multiple Users & Knowledge Bases

While it is important that multiple users be able to make use of the knowledge server, whether it can support multiple users simultaneously is often simply a matter of the operating system it runs under, e.g. does it support shared code? However, whether a single instance

[1] This is not an unreasonable expectation. The X Window System is designed in just this way. It is designed to work across a network, but it is most frequently executed on the same host as the application code which uses it.

of the knowledge server code can share multiple users, or whether multiple instances are needed, isn't really the important point. The important questions are:

- Can the knowledge server support more than one knowledge base? That is, are the procedures and the knowledge base itself distinct?
- Can two users access the same knowledge base simultaneously? If so, then how are updating conflicts handled?
- Can a user combine two knowledge bases? That is, can a user have access to two or more knowledge bases simultaneously?

The last item is probably the most difficult to answer. A collection of *modular* knowledge bases which can be combined with one another would be a useful resource. But a procedure for combining two arbitrary knowledge bases may be impossible to devise.

Communication Protocol

The proposed server/client communication channel is an interprocess communication channel (IPC). This will typically be implemented via a TCP/IP stream, but might also be a UNIX domain IPC if the server and client are executing on the same host.

When knowledge representation systems and applications share the same address space, a reference to a unit in the knowledge base is usually just a pointer within the shared address space. When there is no shared address space, as in the proposed architecture, an alternative must be developed.

A simple solution is to rely on symbolic identifiers. While this is easy to design and implement, it is not particularly efficient since the server has to maintain and use a symbol-to-address mapping for each request from the client. Also, symbolic identifiers are typically larger than numeric identifiers (i.e. that is, they are composed of more bytes).

Another solution is to continue to pass pointers to the data structure, using the address space of the knowledge server, and arrange for a client to locate a particular unit using a naming or indexing scheme.

Extensibility

This is the most difficult design issue that has to be addressed when constructing a knowledge server. While a powerful general purpose reasoner is an important part of an expert system, it is nearly always not enough. Real applications always have a need for special purpose additions to the reasoning routines. In a frame-based representation system, these typically take the form of *attached procedures* written directly in the host language and executed in specified circumstances (e.g. upon the addition or removal of a slot value). A similar situation arises when one wants to directly link the *KR&R* state to the application state, such as reflecting a slot's value in the appearance of a graphic icon.

When the knowledge representation and reasoning components are available as subroutines, it is easy to modify them, or to take advantage of designer provided hooks to add in special purpose procedures. However, when the reasoning routines are located remotely, and may be written in an unknown or inappropriate language, it is more difficult. The questions to be answered in a design are:

- Can the application augment the reasoning process with procedures written in the language of the server?
- Can the application augment the reasoning process with procedures written in the language of the application?

If a user is willing to write special purpose reasoning routines in the same programming language that the server is written in, then there are several traditional techniques techniques for doing this. Most knowledge representation and reasoning tools provide escapes to the implementation programming language specifically to support this need. However, if we assume that the extensions to the reasoning procedures will not be written in the server's programming language, and in fact, can not be executed on the same host that the server is executing on, then other techniques must be devised.

One possible solution is to store the code for the special purpose reasoning routines in the knowledge base and have them returned to the application program at appropriate times to be executed. For some languages, such as Lisp, this may be a reasonable approach, though it is likely to be inefficient. For other languages, it may be difficult or impossible.

A more efficient solution is to have the application implement a *remote procedure call* (RPC) service for the server. The rules of the knowledge representation server can provide the same type of escape to this remote procedure call mechanism that it usually provides for escaping to its native language.

Implementing an RPC service is not necessarily a difficult task. In its simplest form it is just a matter of receiving encoded requests from the Knowledge Server and dispatching them to the appropriate routines for processing. If the application language has the necessary multiprocessing facilities (as do many Lisp systems (e.g. Franz Common Lisp)), an RPC server could be implemented as a separate process using a separate communication channel. Alternatively, it could use the same process and

channel as the application program, along with a communication protocol which is prepared to handle knowledge server queries after each command to the knowledge server. This latter design provides an adequate degree of flexibility, but it may be too slow for intensive use, and it puts a burden on the designer of the application language interface to support this RPC service.

Programming Language Interface

Techniques for accessing the knowledge base server from several languages need to be investigated. For conventional languages, a library of procedure calls will probably suffice. For AI languages like Lisp and Prolog, such libraries will be a good starting point, but when tighter integration with the language is desirable, alternative strategies for accessing the knowledge base can be investigated.

For example, the knowledge base might be accessed as an extension of the Prolog database, or, in Common Lisp, it might return and support a stream datatype to the calling program.

3 Related Work

Although there has been considerable work that is related to the knowledge server idea, none is quite identical with it.

There is a body of work on designing formal interfaces to databases which presupposes a formal database model (the relational model). No such common model exists for knowledge representation. There is work in progress underway to extend formal database models to incorporate more of what we would consider to be the standard set of *KR&R* services [1].

The MCC CYC system [6] is configured with a central knowledge base which many independent CYC processes can access. However, the CYC model assumes that the server and all of the client processes are in Lisp and that each client process loads a copy of the entire CYC knowledge base and the entire CYCL representation and reasoning engine into its image. Thus, the server is used only to provide a central agent to manage the persistence and sharability of the knowledge base and not to actually provide *KR&R* services.

One commercial knowledge representation product, KEE, is available in a network based model [3]. The *PC Host* product offers an expert system shell (KEE) running on a central server machine, supporting users who are running applications on PC class machines attached to the server via a network. However, the division of labor between the KEE server machine and the client machine is significantly different than what we propose. With PC-Host, *all* of the code for the application, both the general purpose and the application specific code, is executed on the server while the application machine simply takes care of the display of output and the low-level handling of input. PC-Host is more like a single process AI application relying on a networked windowing system such as the X Window System, or NeWS, for its low-level user interface.

4 Conclusion

This note describes some preliminary work on the design of a *Knowledge Base Server* - an architecture for delivering knowledge representation and reasoning services to applications. The primary advantages of this approach is the decoupling of the application's general computational needs from it's knowledge-based computation needs. A small application running on a small workstation written in a conventional programming language (e.g. Ada) can access a large AI knowledge base.

Status

We currently have a very simple prototype running and will be implementing a more substantial prototype in the coming year. The current prototype (and future one) uses the Protem system [2] as the knowledge representation and reasoning engine.

Protem is a rule-based knowledge representation system extended by a closely linked frame-based system. As a rule-based system, it offers both forward and backward chaining based on Horn clauses, an integrated justification-based truth maintenance system, and a limited abductive reasoning facility. The frame-based component provides a semantic network which is simple but sophisticated. While it offers the standard *class* and *instance* frames, with *superclass* relations among the frames, it also provides a *metaclass* level for describing the class level. That is, it views classes as instances of other classes.

Protem's strong point is its integration of the two components: its rule-based half and its frame-based half. Neither can operate independently of the other. In particular:

- All assertions and rules are about frames. They either assign values to the attributes of a frame, or assert relations among frames.
- The class/subclass hierarchy of the frame system provides types for the variables of the rule system. This typing is fast and is directly utilized by the unification algorithm.
- Rules and assertions about typed variables are inherited via the subclass links in the frame system in a well defined way. This inheritance is supported by the truth maintenance system which treats them as default values.

Protem is implemented in Common Lisp (supported by the portable implementation of the Common Lisp Object System) and has been run on a variety of machines.

Plans

In the coming year, we plan to implement a more serious prototype knowledge server. We will also implement knowledge server interfaces for Lisp, Prolog and a more conventional language such as Ada or C. In addition, we will be using the knowledge server to provide access to relational databases using a cache-management scheme designed to efficiently connect an knowledge representation system to a conventional database.

References

[1] Michael Brodie. Future intelligent information systems: AI and database technologies working together. In *Readings in Artificial Intelligence and Databases*, Morgan kaufman, San Mateo, 1988.

[2] Rich Fritzson and Tim Finin. *Protem - An Integrated Expert SystemsTool.* Technical Report LBS Technical Memo Number 84, Unisys Paoli Research Center, May 1988.

[3] Intellicorp. *KEE Core Reference Manual.* Intellicorp, 1986.

[4] Thomas S. Kaczmarek, Raymond Bates, and Gabriel Robins. Recent developments in NIKL. In *Proceedings of the* 5^{th} *National Conference on Artificial Intelligence*, pages 978–985, AAAI, August 1986.

[5] Tom Kehler and D. G. Clemenson. An application development system for expert systems. *Systems & Software*, January 1984.

[6] D. Lenat, M. Prakash, and M. Shepherd. CYC: using common sense knowledge to overcome brittleness and knowledge acquistion bottlenecks. *AI Magazine*, 6:65–84, 1986.

[7] Peter F. Patel-Schneider. Small can be beautiful in knowledge representation. In *IEEE Workshop on Prnciples of Knowledge-Based Systems*, pages 11–16, 1984.

[8] Peter F. Patel-Schneider, Ronald J. Brachman, and Hector J. Levesque. ARGON: knowledge representation meets information retrieval. In *Proceedings of the First Conference on Artificial Intelligence Applications.*, pages 280–286, 1984.

[9] Michael Stonebraker and Larry Rowe. The design of POSTGRESS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 340–355, Washington, D.C., May 1986.