An Approach for Creating Distributed Intelligent Systems

Dmitri Soshnikov
Department of Numerical Mathematics and Programming,
Moscow Aviation Institute (Technical University)
Moscow, Russia
shwars@usa.net

Abstract

The paper presents an architecture of a distributed expert system for knowledge sharing across the network, which differs from traditional autonomous agents approach by separating knowledgebase and inference into different components possibly located on different network hosts. By combining inference engine and knowledge provider components in different ways it is possible to combine knowledge on rule-level (on the level of domain knowledge) or on the level of problem state representation. It also allows us to apply several inference strategies to one set of rules, or perform combined inference by using several inference engines in turn.

The article describes domain knowledge and problem state representation most suitable for remote inference, as well as some technical aspects of implementation of such a system using CORBA as remote computation paradigm. Some examples demonstrating possible real-life usages of the remote inference are given.

1. Introduction

Nowadays we observe growing popularity of global computer networks in almost any area of human activity. Internet is now used not only to provide efficient way of transferring data between sites, but also as a huge information source: a global information highway. Moreover, since all computers in the Internet can seamlessly exchange information over any physical distance, all sorts of complex information systems can be built with different parts of the system running on different computers within the network. Such systems, called **distributed systems**, can, for example, provide a framework for **virtual corporation** (a corporation with subdivisions located worldwide, which are tightly integrated to accomplish common tasks), or support complex search and retrieval system over distributed information storage.

A simple distributed system can be based on exchange of certain data between sites, while more complex one can consist of interoperating components called **agents**. An agent is typically an independent software component that is considered with respect to its **environment**, and which performs a certain task by observing the environment and performing **actions** [1]. For example, an agent might regularly check specified web sites and notify you when changes occur, or automatically reply to your mail. By combining agents and making them interoperate cooperatively one can achieve more complex task.

There are cases when certain degree of intelligence is required in the agent's behaviour. Such agents are called "intelligent agents", and usually implement some sort of AI techniques. As an example of such agent consider "consulting agent" which can give you the consultation on certain problem given some initial information. Such an agent would have an encapsulated knowledgebase and inference strategy, i.e. implement an *expert system with remote interface*.

Now suppose we have multiple consulting agents, and we want to combine them to achieve more detailed and broad consultation. In this case the information about the problem (so-called **problem state**) will travel from one agent to another, and each agent will consider the problem and add the additional information it infers. In this case, however, each agent should "understand" problem

Proceedings of the Workshop on Computer Science and Information Technologies CSIT'99
Moscow, Russia, 1999

description in the same way, i.e. there should be consistency in knowledge representation and terminology in the agents' design.

In this scheme each agent has its own inference strategy and the set of rules in the knowledgebase. We can add more flexibility (and more complexity as well) by *not imposing an autonomy restriction*, i.e. by separating knowledgebase and inference. In this architecture, **inference engines** and **knowledgebases** (knowledge sources) are separate entities, and can be located on different network nodes. While this would still allow traditional agent-based approach (by running inference engine and a knowledge source on the same machine), it would also make possible to apply several different inference strategies to the same rule set, or combine different rule sets to be used in one inference process, thus creating more complex structures for knowledge sharing. As we will see later in this article there are cases where this type of architecture can be well applied.

2. Some Approaches to Knowledge Sharing over Computer Networks

Ususual facilities provided by computer networks include several means of transferring **data** between computers and therefore sharing **information** on the network. One of the best examples of information sharing is Internet and the World Wide Web, where huge information resource is made available, organized as a network of linked pages.

However, to share **knowledge** in a similar manner we need to provide certain means to interpret data as a source of knowledge. A simple approach that is used in many expert system shells capable of remote consultation is to select certain **knowledge representation** for domain knowledge, and then transfer the knowledgebase in this representation across the network. Once the domain knowledge has been transferred, the system uses its local copy of a knowledgebase to perform inference locally on a client machine. This approach corresponds to **thick-client model**, and in TCP/IP networks is usually implemented as a Java applet, which downloads the knowledgebase from the server using standard TCP/IP transfer mechanisms (HTTP, Sockets, etc.) As an example of such a system we can name JESS (Java Expert Systems Shell, [2]), which implements a subset of CLIPS [4] (see Fig.1).

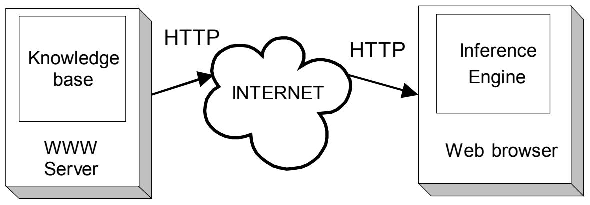

Figure 1 Implementing remote consultation in thick-client model

This model has several disadvantages. First, the representation of domain knowledge is rather large, and transferring it across the network consumes time and network bandwidth. The amount of data that needs to be transferred can be partially reduced by using knowledge representation and inference techniques, which would allow only the rules needed by the inference chain to be actually sent over the network. Another disadvantage is that domain knowledge represents certain intellectual property, and it might be undesirable that it is freely available to clients. This could be avoided by using certain cryptography techniques. At last, the inference is performed on client computer, which requires additional computational resources.

An opposite approach is to transfer not the domain knowledge, but the **problem state**. In this case the initial information about the problem is represented in some way, and then transferred to the server which performs the inference, and returns the new problem state to the caller (refer to Fig.2). This can be achieved by using remote call interface ([3] describes BOW expert systems shell with remote interface, developed at Moscow Aviation Technical University), or through web-based access by calling any traditional expert systems shell (e.g. CLIPS) as CGI application. This approach is very close to the traditional agent architecture, and expert system with a remote interface can be regarded as an agent.

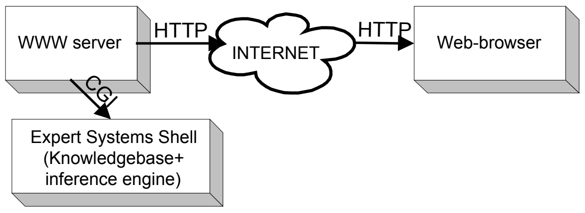

Figure 2 Implementing remote consultation in thin-client model

In both cases knowledge is concentrated in one knowledgebase and then is somehow transmitted between computers. In more complex case, when we want to have several knowledge sources and share knowledge between them (i.e. use the knowledge from all sources in one consultation), we need more complex architecture. As it was mentioned earlier, one way to construct such architecture is to combine agents. This would allow us to integrate knowledge on semantic level, i.e. agents would exchange only problem state representation and not the domain knowledge itself. However, it might be also reasonable to integrate domain knowledge from separate knowledgebases and use it in one inference process. This leads us to the architecture with distributed knowledgebase and inference.

3. Distributed Knowledgebase-Inference Architecture

In the traditional autonomous agent architecture the complex system is constructed from the atomic agents which can, even working individually, carry out a specific task. In the presented approach the system consists of "subatomic" parts called **components**, and several components are required to achieve an intelligent behaviour. This group of components can be regarded as an agent, and such groups can be further combined.

Most important components are **inference engines** and **knowledge providers**. Inference engine is the main reasoning component, which, given the initial data about the problem in the form of so-called problem state, performs the logical inference process using the knowledge from one or more knowledge providers, adding new inferred data into problem state. The inference engine can, in the process of inference of afterwards, pass the problem state to another inference engine, which uses its own or the same knowledge provider. If some more information about the problem is needed in the process of inference, the requests can be made back to the user (or to the calling program). The general structure of the distributed intelligent system is shown on Fig.3.

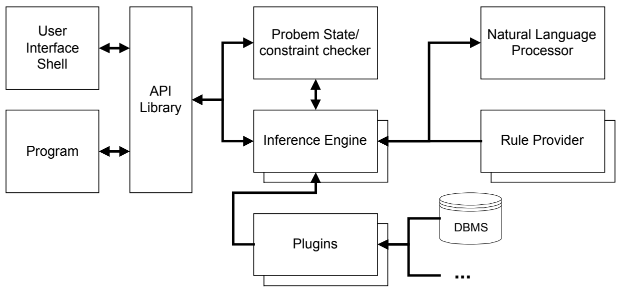

Figure 3 Basic components of distributed knowledgebase-inference model

There are two types of knowledge that need to be represented in the system: domain knowledge and knowledge about the specific problem we need to reason about. Domain knowledge is formed of a general notion of an expert about problem domain, and is independent of a problem being considered. A representation of domain knowledge forms knowledgebase.

Knowledge about specific problem represents the facts that are currently known. We call the set of facts about the problem in a particular moment a problem state. Domain knowledge allows us to infer new facts and add them to the problem state, thus moving to another possible problem state. Thus the inference engine actually performs a search in a set of problem states.

Inference engines may implement different inference strategies, however, to achieve interoperability between all system components, they should all use the same knowledge representation scheme. More precisely, if different knowledge representation schemes are used, only the components with the same representation can be combined to form a reasoning agent. The agents themselves do not exchange domain knowledge but only the problem state, and therefore need to share only the common representation of problem state and not of the domain knowledge.

There could be different levels of abstraction applied to the knowledge representation used to exchange domain knowledge. A good approach would be using standard knowledge exchange language like KQML, which would allow to integrate other KQML-compliant agents into the system. However, one needs to keep in mind that the efficiency of the inference process would depend on the speed the domain knowledge can be transferred to the inference engine. Particularly, it would be useful to have such a representation, where only the part of domain knowledge, relevant to the current inference chain, is obtained from the knowledge provider. The problem state should also be represented in a compact way, since it would be used by different inference engines.

In this respect our approach uses **production rules** as domain knowledge representation, **and attribute-value pairs** to represent problem state. Production rules are simple statements of the form

```
IF <condition>
{ AND | OR <condition> }
THEN <statement>
```

for example

```
IF temperature > 50 AND smoke
THEN house_on_fire
```

This rule, given the evidence that temperature is rather high and there is smoke, will infer the evidence that there is a fire. The evidence itself constitutes problem state, and is represented as a set of **variables** (called **attributes**) having certain **values**. In this example temperature may hold the value of the current temperature, and smoke can be true or false depending on whether there is smoke or not.

There are two basic types of inference used in production expert systems: so-called **forward chaining** (data-driven) inference and **backward chaining** (goal-driven). In the first case, the system considers the initial data and tries to find rules that can be applied (i.e. the ones which conditions are satisfied by the values in the problem state). If a rule is applicable, it adds additional evidence to the problem state. Backward inference, on the contrary, given the goal (the variable whose value is to be found) tries to find the rules which can assign a value to the variable. Then it considers the variables that are needed to satisfy that rule, and tries to find the values of these variables in the same manner. Frequently, in backward chaining inference, if a value of certain variable cannot be determined, the corresponding request is made to the user. For example, starting with no initial data, the backward inferring system can ask for all the data it needs to find the answer by itself. More information on knowledge representation and inference can be found, for example, in [1,4].

One important thing concerning backward inference is that during each step it needs to consider only the rules applicable to find the value of one variable (the current goal), and not the whole knowledgebase. Moreover, if the first such rule can satisfy the goal, no further rules need to be considered. This makes backward chaining perfect to be used in our distributed system, because only few rules that are needed to the inference chain are obtained by the inference engine, while most of the rules in the knowledgebase, irrelevant to the concrete situation, do not need to be transferred.

In some cases it is desirable that forward inference be used. Not to impair the performance, the forward inference engine should be located on the host which has fast network connection to the knowledgebase (or on the same host), because forward inference generally requires all rules to be considered for execution. If the combined inference strategy is to be applied (i.e. after each step of backward inference the forward inference is applied to check if the new facts can be derived), then two inference engines can be located on different hosts, and interoperate. In this case they would both use the same knowledge provider and access the same problem state, and only small portion of the information about additions to the problem state needs to be exchanged between remote hosts.

For different inference engines to operate on the same problem state, the latter is implemented as a separate component which keeps a set of attribute-value pairs and can handle requests to add new pairs. It can also perform some more complex tasks, for example, implement **constraint checks**, disallowing some values to be assigned to the variables. This can be very useful, for example, in medical diagnostic program, when it is known from the evidence that certain diagnosis is incorrect, and we want to check other possibilities. Traditional backward inference architectures do not allow **backtracking**, i.e. facts that have already been inferred cannot be removed from the problem space. Implementing the constraint would prevent the undesired diagnosis from appearing in the problem state at all, and would therefore lead the inference towards different solution.

Making problem state an additional component allows even more complex inference strategy, when more then one inference engine is used to obtain the result. In this case certain rules in the knowledgebase contain references to other inference engines, and, when such rules are executed, the process of inference passes over to another engine. New engine operates on the same problem state, and, when some information is required from the user, directs all questions to the original component that started the consultation.

Apart from the core components the architecture allows easy integration of additional components which share the same knowledge representation and carry out problem-specific tasks. Among such components there may be a natural language processor (to extract facts from natural text or to provide natural-language interface with the user during consultation), DBMS interfaces for getting data out of relational databases, web spiders for extracting information from web pages and more. In the following part we discuss some aspects of implementation of such components.

4. Some words about implementation

While it is always possible to program a distributed system from scratch, there are some traditional methods (we can also say protocols or tools) which facilitate this task. Among them:

- RPC (Remote Procedure Call), which is the oldest protocol introduced in UNIX systems, which allows one process (client) to call a procedure in a remote process (server). RPC hides the implementation details of transferring parameters between machines, and presents the remote procedure to the programmer as though it is a remote procedure.
- RMI (Remote Method Invocation) introduced in Java versions 1.1 and above, which works like RPC, but exposes remote object with several methods to the client program. In a sense it is the object-oriented version of RPC [5].
- DCOM (Distributed Component Object Model) is an extension of Microsoft's COM (Component Object Model) to allow integration of COM Components located on different machines [6].
- CORBA (Common Object Request Broker Architecture) is a standard of distributed computing introduced by a worldwide OMG (Object Management Group) consortium as an alternative to DCOM, promoted by Microsoft. It is also based on component model with well-defined interfaces, through which component methods can be invoked.

In fact, CORBA is more then just a standard for distributed computing, but discussing it is well beyond the scope of this article. Further information about CORBA can be found at OMG web site (http://www.omg.org), or in [7].

The last three models are quite similar, and well suited for development of modern object-oriented distributed systems. The selection of a particular model for implementation is based on many parameters. DCOM is well suited for Windows-Only applications, while RMI is perfect for Java applications because it is part of JDK standard. In the author's view in most of the situations using CORBA has many advantages:

- CORBA is a standard
- CORBA is multilingual: the implementations of CORBA are available for many programming languages including C++, Java, ObjectPascal, Perl and many more
- CORBA is cross-platform and not restricted by any particular operating system

Here we will discuss some of the principles used in implementation of distributed knowledgebase-inference architecture using CORBA.

Each component of the system (knowledge provider, inference engine and so on) is implemented as a CORBA object. For all types of components there are well-defined interfaces, and all particular components implement these interfaces. For example, there is a generic `InferenceEngine` interface, which defines a set of corresponding methods, and all implementations of inference engines (forward reasoning, backward reasoning of combined inference engines) implement this set of functions. In turn all inference engines obtain rules from knowledge providers through generic `KnowledgeProvider` interface, while the actual knowledge provider component can implement different methods for storing rules.

For the actual consultation to take place, there should be at least one inference engine and one rule provider component combined together. To use these components, one need to obtain so-called object references to them, and instantiate the server objects which will actually service requests. To do so, each component has the corresponding **factory object**, which, when asked, will produce the reference to the component itself and prepare the corresponding object implementation on the server side. For example, the interfaces for `RuleProvider` and `RuleProviderFactory` objects in IDL **(Interface Definition Language** similar to C++, which is used to describe interfaces) might look like this:

```
interface RuleProvider
{
void StartTransaction(in string goal);
Rule GetNextRule();
void EndTransaction();
...
}

interface RuleProviderFactory
{
 RuleProvider Create();
 ...
}
```

The server on which `RuleProvider` resides first creates `RuleProviderFactory` object and makes it somehow available to other components (CORBA provides so-called **Naming Service** which facilitates tasks like this). When any other component (like inference engine) needs to use the `RuleProvider`, it calls `the Create()` method of `RuleProviderFactory`, which creates on the server side the `RuleProvider` object, and returns its reference to the calling component, which then can use it. If `Create()` is called one more time, it creates another component, thus allowing multiple components to use the same set of rules. When the `RuleProvider` object is no longer needed, it is destroyed by either explicitly calling corresponding method, or automatically by garbage collector.

When the consultation takes place, some additional information might be required from the user. The inference engine talks to the user through `UserInterface` component, which is created by the program, which uses inference engine (this might be interactive consultation shell, or a software system using remote consultation). The basic function of `UserInterface` is to get request values for attributes which cannot be determined otherwise:

```
interface UserInterface
{
string GetValue(in string value,
      in string_sequence choices);
string Comment(in string s);
}
```

The current problem state is stored as a set of attribute-value pairs, which are accessible through `ProblemState` interface:

```
interface ProblemState
{
 string GetValue(in string value);
 int SetValue(in string attribute,
           in string value);
}
```

The interface of the inference engine is then looks like this:

```
interface InferenceEngine
{
 void Init(UserInterface UI,
ProblemState PS, RuleProvider RP,
...);
 void Execute(in string goal);
 ...
}
```

And of course there are factories for each type of inference engine: Backward reasoning, forward reasoning, etc.

The application that needs the remote consultation obtains references to the factories, instantiates (via corresponding `Create()` methods) `InferenceEngine` and `RuleProvider` objects, and starts the inference. The knowledgebase can contain references to another inference engines: in this case the corresponding engines are created and their `Execute()` method is called, passing the original `ProblemState` and `UserInterface` references to the new inference engine. This process of starting the remote inference can also be encapsulated in a library, which provides API for programs that want to use remote consultation. This library will take care of the details of creating all necessary component instances. Moreover, one can maintain the set of **patterns** typically used in a particular components configuration, stating which components should be used in standard situations, so that API routines can just take the pattern name as a parameter.

5. Examples of Usage

Generally, a process of creating a knowledgebase is a very complex one. A process of creating distributed knowledgebase is even more difficult, because one needs to oversee the way all parts would interact, and keep in mind all rules in different rule providers. However, the whole thing about distributed expert systems is to make a complex system by combining more simple knowledgebases developed independently. It might seem as the presented approach does not provide any means to do that.

We will not discuss here the whole process of developing knowledgebases for distributed intelligent systems. However, there are certain cases where the presented architecture can be successfully used to unify separate knowledgebases to provide more global consultation facilities.

As an example let us consider a computer manufacturer company with several departments in charge of different computer parts, for which we want to create an expert system to help user find the computer configuration (a set of components needed to assemble a computer) and its price depending on user's needs. In this case each department could have a separate knowledgebase of products, and the central knowledgebase would consult each one in turn to find the applicable products and their prices. For example, if the user wants server solution, then the central knowledgebase would look for devices for fast hard disk transfer, and the corresponding department knowledgebases would suggest fastest products available. Using distributed approach has another advantage: separate knowledgebases of each department could be used not only as a part of consultation program, but also in other applications. For example, if several firms use the same approach, we can easily set up an application to compare available hardware solutions.

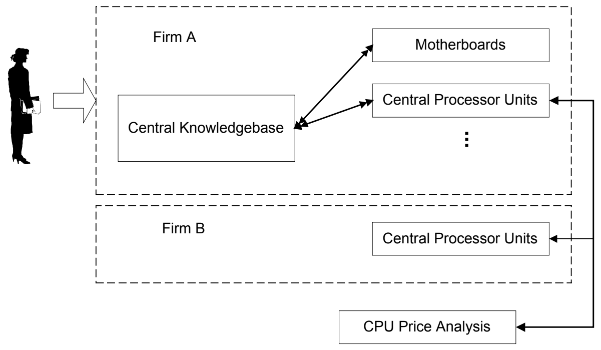

Figure 4 An example of distributed system with separate knowledge domains

In this example the whole distributed knowledgebase is relatively easy to construct because separate knowledgebases can be *considered independently*. The reason for it is that all knowledgebases have **different knowledge domains**, i.e. they reason about different things and in different terms. Only the terms they use to describe the initial problem state and the goals they can reason for need to be known to others to use the knowledgebase. In a sense the separate knowledgebases are similar to components: they have well-defined interface attributes and other attributes which they encapsulate and use only for internal reasoning.

In case we want to compare inference results of different knowledgebases (as mentioned in the example above), we deal with the opposite situation, where knowledge domains are the same. In order to avoid interference of terms from different knowledgebases, a bit more complicated approach should be used. The master program should set up different problem states for different knowledgebases, and then inference could be performed parallely.

The examples considered are quite typical for real-life situations, where information flow has hierarchical structure, and therefore the problem can be separated into distinct knowledge domains. Medical diagnosis program in a hospital could be another example, where each doctor would maintain his own knowledgebase for diagnosis, and, combining them we would be able to obtain a complete diagnosis based on patient's complaints. And we can also perform consultation with different specialists' knowledgebases and compare the results. Moreover, we can set up more complex structure, where the diagnosis is performed by several groups of individual knowledgebases, each group providing complete diagnosis, and the results of all groups are then compared or combined. Of course, for complex diagnostic problems probabilistic approach would be highly desirable, which would require major improvements to the relatively simple architecture described here.

6. Conclusion

As it can be seen in the examples above, there are many different applications to the described approach. The main advantage of the architecture is the additional flexibility comparing to traditional agent-based approach. By separating knowledgebase and inference it is possible to combine knowledge on different levels of abstraction, from simple rule merging to the complex combination of rules and several types of inference. By using CORBA as an infrastructure for distributed computing it is possible to implement and use system components in several programming languages and on different platforms, across virtually any network. The architecture described can be useful in implementing intelligence in distributed information systems, which are widely used, for example, in virtual corporations, thus bringing the advantages of distributed computing to the new level.

References

1. Russel S., Norvig P. "Artificial Intelligence: A Modern Approach". Prentice Hall, 1994.
2. JESS web page: http://herzberg.ca.sandia.gov/jess/
3. Zaitsev V.E., Lukashevich S.Y., Soshnikov D.V., Terlekchiev K.R., Raldugin V.N. "Remote API to the BOW expert systems shell". Moscow Aviation Technical University, 1998. (In Print)
4. Giarratano J., Riley G. "Expert Systems: principles and programming". PWS Publishing Company, Boston, 1994.
5. Weber J. "Special Edition: Using Java". QUE Corporation, 1996.
6. Rogerson D. "Inside COM". Microsoft Press, 1997
7. Brjukhov D.O., Zadorozhniy V.I., Kalinichenko L.A., Kuroshev M.Y., Shumilov S.S. "Interoperable information systems: architecrure and technology". *DBMS*, 1995; No. 4. (In Russian)