IMPACT: The Interactive Maryland Platform for Agents Collaborating Together*

Khaled Arisha **Sarit Kraus** **Fatma Ozcan** **Robert Ross**
V.S. Subrahmanian †

Abstract

In this paper, we describe a platform called IMPACT to support multiagent interactions. The platform provides a set of servers (yellow pages, thesaurus, registration, type and interface) that facilitate agent interoperability in an application independent manner. In IMPACT, agents have an associated set of service descriptions, specifying the services they provide. We develop an HTML-like language for such service descriptions. When an agent wishes to identify another agent that provides a service, the requested service must be matched, using metric approach, against existing service descriptions. We provide a formal framework within which this may be done, and develop algorithms to compute the k nearest matches, as well as all matches within a given distance from the requested service. We report on experiments evaluating our algorithms with large data sets.

Subject Area: Agent models and architectures

1 Introduction

During the last few years, there has been tremendous interest in the area of *Software Agents*. The Webster Dictionary defines an *agent* as:

1. "One who exerts power, or has the power to act; an actor."
2. "One who acts for, or in the place of, another, by authority from him; one intrusted with the business of another; a substitute; a deputy; a factor."
3. "An active power or cause; that which has the power to produce an effect; as, a physical, chemical, or medicinal agent; as, heat is a powerful agent. "

Consequently, a simplistic definition of a "software agent" is any software program that satisfies any of the above criteria. As these criteria are very broad, almost any program which accomplishes something (even one that merely changes the value of a register in memory) is an agent as it produces an effect.

In this paper, we focus on *sets of interacting agents* and ask ourselves the following questions:

*This work was supported by the Army Research Office under Grants DAAH-04-95-10174, DAAH-04-96-10297, and DAAH04-96-1-0398, by the Army Research Laboratory under contract number DAAL01-97-K0135, and by an NSF Young Investigator award IRI-93-57756.

†Dept. of Computer Science, Institute for Advanced Computer Studies, and Institute for Systems Research, Bar-Ilan University, Ramat Gan Israel, and University of Maryland, College Park, Maryland 20742. E-Mail: { arisha,sarit,fatma,robross,vs}@cs.umd.edu

1. What properties does a program need to have in order to meaningfully and efficiently interact with other (similar) programs ?

2. If a program P does not have the aforementioned properties, is there a way to expand it (without delving into the internal code of P) so that the resulting program does have the desired properties ?

3. How should we specify the circumstances under which one agent seeks support from another "helper" agent and how are these "helper" agents identified?

4. What underlying software layer is needed to support interactions between multiple programs possessing the properties alluded to in (1) above?

The above questions admit a multiplicity of answers, each of which will potentially lead to different implementation strategies. In this paper, we will present one set of answers to the above questions. In particular:

1. We will use the word "agent" to denote any program P that has the ability to:
   (a) express the services that P can offer to other agents;
   (b) express what inputs it needs from other agents to provide these services;
   (c) understand (perhaps with help from the underlying software layer) the requests and/or messages it receives.

2. We provide an HTML-like (called *Agent Service Description Language*) language within which agents may specify the services they provide, as well as the inputs they require for each service, and the outputs provided. This HTML-like description may be appended as "glue" on top of a program.

3. We develop an extension of this HTML-like language (*Agent Cooperation Language*) that allows the (human) owner of an agent to extend the range of services provided by the agent possibly by cooperating with another agent or agents.

4. We specify a set of servers including those briefly listed below that form the underlying software layer for multiagent interactions.
   (a) Yellow Pages Server: This server takes as input, a request to find an agent that provides a service, and returns as output, a ranked list of such agents.
   (b) Registration Server: This server is used when a new agent is introduced into the existing set of agents. It indexes the services offered by different agents, and is used by the yellow pages agent to retrieve agents providing a specified service.
   (c) Type Server: This server maintains an ontology of types, both standard data types like `reals,integers, char`, as well as semantic types like `country,currency`, etc.

(d) Thesaurus Server: This server provides the services usually provided by a thesaurus.

(e) Human Interface: This interface allows a human to access all the above.

5. We describe a software system called **IMPACT** (short for "Interactive Maryland Platform for Agents Collaborating Together") that implements the above theory, and we describe some experiments on agent interaction, as well as some applications that we have built using **IMPACT**.

2 Agent Service Description Language

Any agent A created by a programmer consists of a piece of code implementing a set of services, together with a *description* of what these services are. While different languages may be used to implement the code of the agent, we require that all service descriptions be provided in a single HTML-like language that we now specify. This language is now described using a series of definitions. In the sequel, we will use the expressions "verb" and "noun" in their usual English language sense.

Intuitively, a specification of a single service consists of;

- **Service Name:** This is a verb-noun(noun) expression describing the service. For example, `plans:travel()` is a service name specifying a service for travel planning. Similarly, `sell:car(Japanese)` is a service name describing a service that sells Japanese cars.
- **Inputs:** Services assume that the users of the service will provide zero or more inputs. The service description must include a specification of what inputs are expected, and which of these inputs are mandatory. This specification must provide an "English" name for each input, as well as a semantic type for that input. For example, `destination:city` specifies that we have an input called `destination` of type `city` (which could be an enumerated type).
- **Outputs:** Each service must specify the outputs that it provides and each output is specified in the same way as an input.
- **Attributes:** In addition, services may have attributes associated with them such as cost (for using the service), average time of responses to requests for that service, etc.

In order to define service names, we first introduce the concept of a noun-term.

Definition 2.1 (Noun Term) *If n_1, n_2 are nouns (where n_2, but not n_1, may also be the empty string) in English, then $n_1(n_2)$ is a* noun term. *In this case, n_1 is called the* root *of this noun term. When n_2 is empty, we will often abuse notation and write n_1 instead of $n_1()$.*

For examples, *car(Japanese)*, *tickets(plane)* are noun terms. Noun terms by themselves mean nothing. However, noun terms considered jointly with a verb define a service name.

Definition 2.2 (Service Name) *If v is a verb in English, and $n_1(n_2)$ is a noun term, then both v and $v : n_1(n_2)$ are* service names.

Thus, the reader will notice that a verb by itself can constitute a service, e.g. the verb `search` may be a service name. Similarly, `sell:car(Japanese)` defines a service name as well. Each service expects the client which is requesting the service to provide certain inputs having a specified type. Before defining input specifications, we therefore need to formally define types.

Definition 2.3 (Type/Type Hierarchy) *A* type *τ is a set whose elements are called "values" of τ. The pair $(\mathcal{T}, \leq)$ is called a* type hierarchy *if $\mathcal{T}$ is a set of types, and $\leq$ is a partial ordering on $\mathcal{T}$.*

Definition 2.4 (Type Variable) *Associated with any type hierarchy $(\mathcal{T}, \leq)$, is a set $V_{\mathcal{T}}$ of symbols called* type variables.

Intuitively, a type variable ranges over the values of a given type. For instance, `Author` may be a type variable ranging over strings. When specifying the inputs required to invoke a service, we need to specify variables and their associated types. This is done in the usual way, as defined below.

Definition 2.5 (Item) *If s is a variable ranging over objects of type τ, then $s : \tau$ is called an* item.

For example, `Author:String`, `Document:Ascii_file`, and `Addr:Netaddress` are all valid items, if one assumes that the types "String", "Ascii_file" and "Netaddress" are all well defined. As is common in most imperative programming languages, the syntactic object $s : \tau$ may be read as saying "The variable s may assume values drawn from the type τ."

Most services require zero, one, or more inputs. Some of these inputs are mandatory (i.e. the service cannot or will not honor a request for the service if these inputs are not provided), while others are discretionary (the service would like them, but does not insist they be provided). For example, the `sell:car(Japanese)` service may require that the `model` and `maxcost` fields be filled, but may not require a `sunroof` field to be filled in (though the user may specify that he wants or does not want a sunroof if he so desires). This is captured in the following definition.

Definition 2.6 (Item Atom) *If $s : \tau$ is an item, then $\langle \mathtt{I}\rangle \mathtt{s} : \tau \langle \backslash \mathtt{I}\rangle$ (resp. $\langle \mathtt{MI}\rangle \mathtt{s} : \tau \langle \backslash \mathtt{MI}\rangle$) is called an* input (resp. mandatory input) *item atom, and $\langle \mathtt{O}\rangle \mathtt{s} : \tau \langle \backslash \mathtt{O}\rangle$ is called an* output *item atom.*

Each input item is either *mandatory* or not. For example $\langle \mathtt{MI}\rangle$model:japanese_car$\langle \backslash \mathtt{MI}\rangle$ is a mandatory input item atom, while $\langle \mathtt{I}\rangle$sunroof:boolean$\langle \backslash \mathtt{I}\rangle$ is a non-mandatory input item atom. Similarly, $\langle \mathtt{O}\rangle$cost:real$\langle \backslash \mathtt{O}\rangle$, $\langle \mathtt{O}\rangle$specs:car_spec_record$\langle \backslash \mathtt{O}\rangle$ and $\langle \mathtt{O}\rangle$financing_plan:finance_record$\langle \backslash \mathtt{O}\rangle$ are all valid output item atoms.

Definition 2.7 (Service Description) *If* `sn` *is a service name, and $i_1, \ldots, i_n$ are input item atoms, $mi_1, \ldots, mi_k$ are mandatory input item atoms, and $o_1, \ldots, o_r$ are output item atoms, then*

⟨S⟩

sn

$mi_1, \ldots, mi_k$

$i_1, \ldots, i_n$

$o_1, \ldots, o_r$

⟨\S⟩

is called a service description.

For example, Travelocity is a well known web site (`www.travelocity.com`) providing travel services. Using our service description language, we may describe Travelocity's route serive (to find a route between two points on a map) as follows:

⟨S⟩

find:route

⟨MI⟩	from_streetnumber:integer ⟨\MI⟩
⟨MI⟩	from_street:string ⟨\MI⟩
⟨MI⟩	from_city:city ⟨\MI⟩
⟨MI⟩	from_state:us_states ⟨\MI⟩
⟨MI⟩	to_streetnumber:integer ⟨\MI⟩
⟨MI⟩	to_street:string ⟨\MI⟩
⟨MI⟩	to_city:city ⟨\MI⟩
⟨MI⟩	to_state:us_states ⟨\MI⟩
⟨O⟩	map:file ⟨\O⟩

⟨\S⟩

The full version of this paper shows how a wide sample of well known services available through the World Wide Web may be described using our service description language. These examples run a wide gamut, ranging from travel services, bibliographic search services, and mortgage services.

3 IMPACT Architecture: Design and Implementation

The **IMPACT** architecture shown in Figure 1 supports the interactions that occur between a set of agents that may be located at geographically dispersed sites on a network. Agents are shown in Figure 1 as a white box (representing the code that implements the agent), surrounded by a yellow "wrapper" which contains the agent service description. As we have already discussed these wrappers in great detail in Section 2 above, we will not go into this in greater detail here.

Instead, we will concentrate on the **IMPACT** software layer that provides the infrastructure upon which different **IMPACT** agents may interact. The servers constituting this layer are shown as pink rectangles in Figure 1. This layer does not necessarily reside at one location on the network – multiple copies of it may be replicated and scattered across the network. Each copy of the **IMPACT** software layer also has a synchronization component that updates other copies when the software layer itself gets modified. The synchronization component is shown in green in Figure 1.

Before discussing the individual components of the **IMPACT** architecture, we need to understand how service descriptions are used. An agent A specifies descriptions of the services it

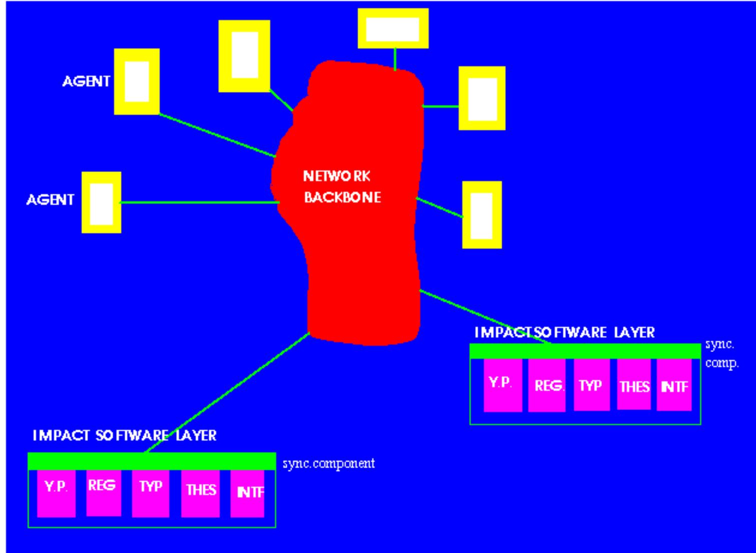

Figure 1: **IMPACT** Architecture

provides so that other agents or humans can access those services. Furthermore, an agent B may use terminology different from agent A to describe a service that it is seeking. For example, agent B may want to find agents that provide cars. Agent A may specify that it offers the service `sell:car(Japanese)`, but agent B may be searching for a service specified as `provide:car`. Here, different terms are used to denote a similar service – however, determining that these two syntactically different expressions are semantically similar is nontrivial. The **IMPACT** architecture maintains a set of data structures (together with traversal algorithms) that facilitates such retrievals.

1. First, a Yellow Pages server maintains two weighted hierarchies – a *verb* hierarchy and a *noun* hierarchy. The nodes in the verb hierarchy are sets of synonym verbs, while the nodes in the noun hierarchy are sets of synonymous noun-terms (exactly what it takes for two noun terms to be synonymous will be described later). If node N is an ancestor of node N' in either of these hierarchies, it means that the labels of N are more general than those of N'. For example, the verb `provide` is more general than `sell`, and the noun `car` is more general than `car(Japanese)`. Weights on the edges denote degrees of dissimilarity – the larger the weight, the less similar the nodes involved. The Yellow Pages Server implements a similarity measure through which the user (or an agent) may identify which agents provide (a service similar to) a desired service. The Yellow Pages Server also implements a variety of retrieval algorithms to compute these similarities efficiently.

2. New agents, when created and ready for release, are registered with a Registration Server. This registration server provides an *interface* through which the data structures maintained by the Yellow Pages Server may be updated to reflect the services provided by the new agent. It also provides valuable browsing services which may be used by the owner of the new agent

to see what terms exist currently in the verb and noun hierarchies and how can be used to characterize the services of the new agent. This allows the owner to modify the service description of its agent to take into account, existing terminology rather than expand the existing vocabulary needlessly.

3. The browsing supported by the Registration Server includes not only the noun and verb hierarchies, but also access to other servers such as a Type Server that maintains type information as a hierarchy (e.g. the type "city" may be a subtype of the type "place") which is used to specify the τ component of an "item" $s : \tau$ as described earlier. Similarly, a Thesaurus Server allows the owner of a new agent to browse a thesaurus and find words similar to the ones he is using to describe services.

4. Last but not least, when the owner of an agent wishes to register the agent and commits to such a registration, the **IMPACT** software layer he accesses the registration service from must notify other mirror, replicated copies of this fact so that consistency across different versions of the **IMPACT** software layer is maintained. This is done by the **IMPACT** software layer.

3.1 Yellow Pages Server

In this section, we will first describe the data structures maintained by the Yellow Pages Server. There are three such structures – two hierarchies that are described in Section 3.1.1 below, and an agent table described in Section 3.1.2. Agents may ask the yellow pages server to identify other agents that provide certain services. These requests may take several different forms which we will describe in this section. Algorithms to process these requests will be presented in Sections 3.1.3 and 3.1.4.

3.1.1 Term Hierarchies

In this section, we will provide a general definition of a Term-Hierarchy. Both the noun hierarchy and verb hierarchy alluded to earlier in this paper will turn out to be special cases of the general concept of a term-hierarchy. We start by assuming that there is a set Σ whose elements are called *terms*, and an equivalence relation $\sim$ on Σ. Intuitively, one can think of Σ as a set of words, and the relation $\sim$ as a representation of synonymity. For example, Σ could be a set of verbs, and $v_1 \sim v_2$ may indicate that verbs v_1, v_2 are synonymous.

Definition 3.1 (Σ-node) *A* Σ-node *is any subset $N \subseteq \Sigma$ that is closed under $\sim$, i.e.*

1. $x \in N \;\&\; y \sim x \Rightarrow y \in N$.

2. $x, y \in N \Rightarrow x \sim y$.

In other words, Σ-nodes *are equivalence classes of* Σ.

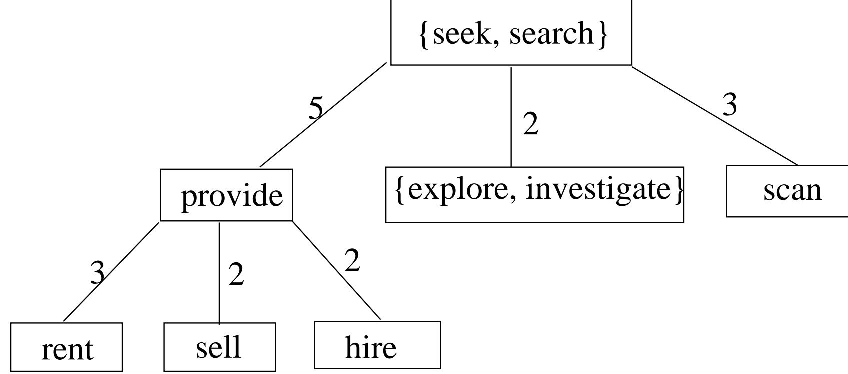

Figure 2: An example verb hierarchy

For example, suppose Σ_1 consists of just a few terms given by:

$$\Sigma_1 = \{explore, investigate\}.$$

If we use the symbol $\sim_1$ to denote the ordinary notion of equivalence amongst these terms, the following are Σ_1-nodes:

$$N_1 = \{car, automobile\}.$$
$$N_2 = \{truck, pickup\}.$$

Definition 3.2 (Σ-Hierarchy) *A Σ-Hierarchy is a weighted, directed acyclic graph $\mathcal{SH} = (T, E, \wp)$ such that:*

1. *each vertex of T is a Σ-node;*
2. *if $t_1, t_2 \in T$, then t_1 and t_2 are disjoint;*
3. *$\wp$ is a mapping from E to $\mathbf{Z}^+$ indicating a positive distance between two neighboring vertices*[1]*.*

When $\mathcal{SH}$ is fixed, we use triples of the form (t_1, t_2, d) to denote an edge from t_1 to t_2 (in $\mathcal{SH}$) having $\wp(t_1, t_2) = d$.

Figures 2 and 3 show a sample verb and noun-term hierarchy respectively. The fact that the edge between node $\{seek, search\}$ and $\{provide\}$ has a weight of 5, while the edge between $\{seek, search\}$ and $\{scan\}$ has a weight of 3, indicates that $\{scan\}$ is more similar to $\{seek, search\}$ than $\{provide\}$.

Definition 3.3 (Σ-Path) *A Σ-Path between two nodes $t, t' \in T$ in the Σ-Hierarchy $\mathcal{SH} = (T, E, \wp)$ is a sequence $t_1, \ldots, t_n$ such that: $t_1 = t, t_n = t'$ and for all $1 \leq i < n$, $(t_i, t_{i+1}) \in E$. The* length *of such a path is $(n-1)$. The* cost *of this path is the sum of the weights of the edges along the path.*

[1] We do not require $\wp$ to satisfy any metric axioms at this point in time.

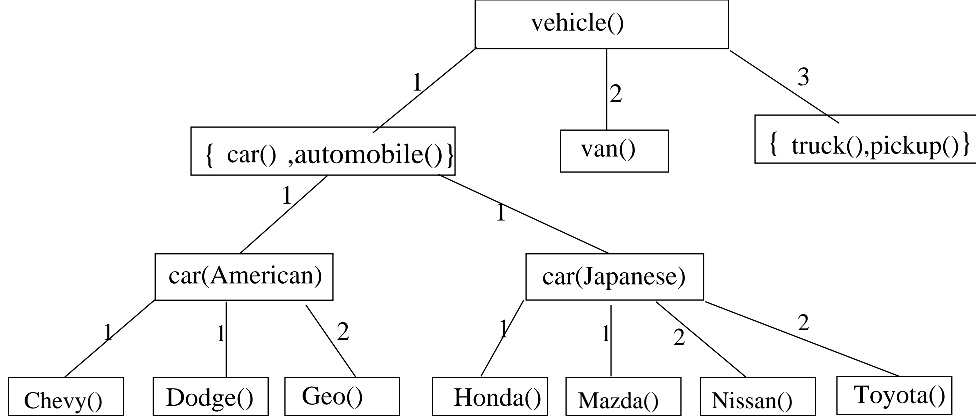

Figure 3: An example noun hierarchy

We are now ready to define the notion of distance between two nodes in a hierarchy.

Definition 3.4 (Distance) *Consider a Σ-hierarchy, $\mathcal{SH} = (T, E, \wp)$, and suppose $w_1, w_2 \in \Sigma$ are terms. Then the* distance *between w_1, w_2, denoted $d_{\mathcal{SH}}(w_1, w_2)$ w.r.t. $\mathcal{SH}$ is defined as:*

$$d_{\mathcal{SH}}(w_1, w_2) = \begin{cases} 0 & \textit{if there exists } t \in T \textit{ such that } \{w_1, w_2\} \subseteq t \\ cost(p_{min}) & \textit{if there is a } \Sigma\textit{-path between } w_1, w_2 \textit{ and } p_{min} \textit{ is a least cost such path} \\ \infty & \textit{otherwise} \end{cases}$$

It is easy to see that given any Σ-hierarchy, $\mathcal{SH} = (T, E, \wp)$, the distance function, $d_{\mathcal{SH}}$ induced by it is well defined and satisfies the triangle inequality.

As the reader has already noted, service names in **IMPACT** are of the form *verb:noun_term*. Typically, this leads to two hierarchies, one for verbs and one for noun-terms, which we will denote by $\Sigma_v = (T_v, E_v, \wp_v)$ and $\Sigma_{nt} = (T_{nt}, E_{nt}, \wp_{nt})$, respectively. When human beings or agents access the Yellow Pages server, they specify a service $v_Q : nt_Q$, and make one of two requests:

k-Nearest Neighbor Request: Find the k "nearest" pairs (v, nt) in the hierarchy such that there exists an agent who provides that service and identify that agent.

δ-Range Search Find all pairs (v, nt) within a specified "distance" δ such that there is an agent who provides that service, and identify that agent.

However, thus far, we have only defined distances on Σ_v and Σ_{nt} individually – no notion of distance has been proposed thus far between *pairs* of verbs and noun-terms. We now define what it means to *combine* two distance functions on two hierarchies.

Definition 3.5 (Composite Distance Function) *Suppose we have two different hierarchies $\Sigma_1 = (T_1, E_1, \wp_1)$ and $\Sigma_2 = (T_2, E_2, \wp_2)$. Let d_1, d_2 be the distance functions induced by Σ_1, Σ_2, respectively. Consider two pairs of words, $(w_1, w_1'), (w_2, w_2') \in \Sigma_1 \times \Sigma_2$. A* composite distance *function cd, is any mapping from $(\Sigma_1 \times \Sigma_2) \times (\Sigma_1 \times \Sigma_2)$ to $\mathbf{Z}^+$ such that:*

- $cd((w_1, w'_1), (w_2, w'_2)) = cd((w_2, w'_2), (w_1, w'_1))$.
- $cd((w_1, w'_1), (w_1, w'_1)) = 0$.
- *If* $d_1(w_1, w_2) \leq d_1(w_1, w_3)$ *then* $cd((w_1, w'_1), (w_2, w'_2)) \leq cd((w_1, w'_1), (w_3, w'_2))$.
- *If* $d_2(w'_1, w'_2) \leq d_2(w'_1, w'_3)$ *then* $cd((w_1, w'_1), (w_2, w'_2)) \leq cd((w_1, w'_1), (w_2, w'_3))$.
- $cd((w_1, w'_1), (w_3, w'_3)) \leq cd((w_1, w'_1), (w_2, w'_2)) + cd((w_2, w'_2), (w_3, w'_3))$.

The reader may wonder whether that last item in the above definition is redundant. It turns out that it is not redundant (but due to space restrictions, we do not present a proof of this here).

3.1.2 Agent Table

The agent table is a relational table containing three attributes – Verb, NounTerm and Agent. The table has NounTerm as its primary key, and Verb as the secondary key. The last column, "Agent", is an agent id. Intuitively, suppose the tuple

$$(rent, car(Japanese), agent1)$$

is in this table. This means that agent1 provides the service (rent,car(Japanese)). The table below shows an example agent table. Notice that in this table, there are two agents that provide the service (rent,car(Japanese)).

Verb	Noun Term	Agent
rent	car(Japanese)	agent1
rent	car(American)	agent3
rent	vehicle()	agent6
hire	Honda()	agent4
rent	car(Japanese)	agent2
sell	Mazda()	agent2
rent	vehicle()	agent3

In **IMPACT**, the agent table is implemented as an Oracle relation. We have implemented an operation called *search_agent_table* which takes three arguments – a verb V, a noun term NT, and in integer K. This function searches the agent table to find at most K agents that provide the exact service (V, NT). In fact, this function first executes the SQL query:

SELECT Agents
FROM AgentTable
WHERE Verb = V **AND** NounTerm=NT.

This SQL query returns a set of agents. If this set has K or more elements, then the *search_agent_table* function returns K of these, otherwise (i.e. if this set has less than K elements) it returns the entire set.

3.1.3 k-Nearest Neighbor Request Algorithm

In this section, we will develop an algorithm *find_nn* to solve the k-nearest neighbor problem. Given a pair (v, nt) specifying a desired service, this algorithm will return a set of k agents that provide the most closely matching services. The most closely matching services are determined by examining:

- The verb hierarchy,
- the noun hierarchy, and
- the thesaurus.

Closeness between (v, nt) and another pair (v', nt') is determined using the distance functions associated with the verb and noun-term hierarchies, together with a composite distance function *cd* specified by the agent invoking the *find_nn* algorithm. In addition to using the above structures and the agent table, our algorithm uses the following internal data structures and/or subroutines:

1. *Todo:* This is a list of verb-noun term pairs maintained in increasing order of distance from the verb-noun term pair that is requested in an explicit call (either the initial call or a recursive call) to the *find_nn* function. This list is not necessarily complete.

2. *ANSTABLE:* This is a table consisting of at most k entries (k being the number of agents requested). At any given point in time during execution of the *find_nn* algorithm, *ANSTABLE* will contain the best answers found thus far.

3. *num_ans*: This function merely keeps track of the number of answers in *ANSTABLE*.

4. *next_nbr:* This function takes as input, the list *Todo* mentioned above, and a pair (V, NT). Intuitively, *Todo* consists of verb noun-term pairs that have not yet been relaxed. Suppose (V_1, NT_1) is the first verb noun term pair in this list. The *candidate-relaxations*, $cr(V_1, NT_1)$ of (V_1, NT_1) are defined as follows:

   - If V' is an immediate neighbor of V_1 in the verb hierarchy, then (V', NT_1) is in $cr(V_1, NT_1)$.
   - If NT' is an immediate neighbor of NT_1 in the noun-term hierarchy, then (V_1, NT') is in $cr(V_1, NT_1)$.

   The function *next_nbr* removes (V_1, NT_1) from the *Todo* List, and then inserts all members of $cr(V_1, NT_1)$ into the *Todo* list while maintaining the property that the *Todo*-list is in ascending order of the distance from (V, NT). The function *next_nbr* returns as output, the first member of the *Todo* list that has not been examined before.

5. *relax_thesaurus:* This function is similar to accesses the thesaurus and the verb/noun term hierarchies. When invoked with the input pair (V, NT), it does the following:

   - If V is not in the verb hierarchy, then it searches the thesaurus for a verb V' that is in the verb hierarchy and is synonymous with V, and sets V to V'.

- If NT is not in the noun-term hierarchy, then it searches the thesaurus for a noun-term NT' that is in the noun-term hierarchy and is synonymous with NT and sets NT to NT'.

If either V or NT after this step is still not in the verb or noun-term hierarchy, respectively, then it returns a pair of the form (♯ERROR,-). Otherwise, it returns the pair (V, NT).

For example, we may have a thesaurus which contains (amongst other words), the following:

Word	Synonyms
explore	inquire,examine,probe,investigate
vehicle	carrier, carriage,van,wagon,car,track
car(Japanese)	Honda,Nissan,Mazda,Toyota
scan	browse,glance,skim
provide	supply, hand-over
seek	search, lookup
rent	hire,lease
car(American)	Chevy, Neon, Dodge

```
Algorithm 1  find_nn(V:verb,NT:noun-term,K:integer)

  done = false;
  V_ini = V; NT_ini = NT;
  create(Todo, V, NT);
  ClosedList = NIL;
  ANSTABLE = ∅;
  if V ∈ Σ_v & NT ∈ Σ_nt then
  {
    SOL = search_agent_table(V,NT,K);
    while ¬done do
    {
      insert((V,NT),ClosedList);
      insert(SOL,ANSTABLE);
      n = num_ans(ANSTABLE);
      if n ≥K then done = true
      else
      {
        (V',NT') = next_nbr(Todo);
        if V'= ♯ERROR then done = true
        else
        {
          V = V'; NT = NT';
          SOL = search_agent_table(V,NT,K-n);
        } (⋆ end inner if ⋆)
      } (⋆ end middle if ⋆)
    } (⋆ end while ⋆)
  } (⋆ end outer if ⋆)
  else
  { (⋆ search thesaurus ⋆)
    (V',NT') = relax_thesaurus(V,NT);
    if V'= ♯ERROR then return ERROR
    else find_nn(V',NT',K)
  }
  return ANSTABLE;
end
```

Example 3.1 Consider the verb and noun-term hierarchies given in Figures 2 and 3 respectively and the agent table shown above and the thesaurus shown above. Furthermore, suppose the composite distance function, *cd*, is addition.

Consider the call *find_nn(rent, car(),2)* which requests 2 agents that rent cars. Initially, V_{ini} = rent and NT_{ini} = car(). The list *Todo* contains (rent,car()) when first created. Since rent $\in \Sigma_v$ and car() $\in \Sigma_{nt}$, we execute *search_agent_table (rent, car(),2)*. This returns the empty set since there is no agent providing the service `rent:car()`. (rent,car()) is now relaxed by calling the function *next_nbr* which returns (rent, vehicle()). The list *Todo* now contains the {*(rent,car(Japanese)),1), (rent, car(American)),1),(provide, car()), 3)* } The call *search_agent_table (rent, vehicle(),2)* will return agent6 which provides `rent:vehicle()` service. This result is inserted into *ANSTABLE* with its composite distance function value of 1. As we still need to find another answer, the *next_nbr* function is called again. This call returns the (rent,car(Japanese)) verb, noun-term pair and the call *search_agent_table (rent, car(Japanese),1)* results in agent1 which provides `rent:car(Japanese)`.

Note that although agent2 also provides the `rent:car(Japanese)` service, *search_agent_table* function only returns agent1 because only one is requested. Now that there are two answers in the *ANSTABLE*, the *find_nn* function terminates by returning $\{(\text{agent6},1), (\text{agent1},1)\}$.

3.1.4 Range Neighbor Search Algorithm

The range search algorithm below allows the IMPACT server to answer queries of the form "Find all the agents that provide a service (V', NT') which is within a distance D of a requested service (V, NT)."

In the algorithm below, *Todo* is a list of nodes to be processed. The algorithm has two steps. The first step is the **while** loop. This step finds all pairs $(V^\star, NT^\star)$ that are within the specified distance D from (V, NT). This step uses a function called *Expand* that behaves as follows. $Expand(V, NT, V', NT', D)$ first computes the set $\{(V^\sharp, NT^\sharp) |$ the distance between $(V^\sharp, NT^\sharp)$ and (V, NT) is less than or equal to D and $(V^\sharp, NT^\sharp) \in cr(V', NT')$ and $(V^\sharp, NT^\sharp)$ is not in *RelaxList* $\}$. It then adds this set to the *Todo* set and returns the result.

The second step executes a select operation on the Agent Table, finding all agents that offer any of the service names identified in the first step.

Algorithm 2 *range(V:verb,NT:noun-term,D:real)*

$RelaxList = NIL;$
$Todo = \{(V, NT)\};$
while $Todo \neq NIL$ **do**
{
$(V', NT') =$ *first element of Todo;*
$RelaxList = RelaxList \cup \{(V', NT')\};$
$Todo = (Todo - \{(V', NT')\}) \cup Expand(V, NT, V', NT', D);$
};
Return $\bigcup_{(V',NT') \in RelaxList} (\pi_{Agents}(\sigma_{Verb=V' \,\&\, NounTerm=NT'}(AgentTable)));$
end

3.2 Registration Server

Whenever a new agent is created by a programmer, this new agent must be registered with the IMPACT server. In particular, the new agent A must have the following associated specifications:

- A set $\mathsf{Svc}(A)$ of services that are provided by agent A;
- For each service $s \in \mathsf{Svc}(A)$, a description of the inputs required from clients wishing to make use of that service, together with a description of the outputs provided.

The registration server provides the following facilities to make this possible.

1. **Hierarchy Search and Browsing:** First, it contains an interface through which the user can browse the existing verb and noun-term hierarchies to see if there are words in these hierarchies that can be used to describe the services of agent A. The registration server also

supports querying the hierarchy, i.e. the user can ask if the word `honda` is in the noun-term hierarchy, and if so, an appropriate part of the noun-term hierarchy above, below, and including `honda` is displayed to the user.

2. **Hierarchy Updates:** Once this is done and the user has determined exactly how he wants to specify his agent's services, the registration server inserts any new words introduced by the user (which were not previously in the hierarchies) into the hierarchies, in consultation with the user. Only authorized users (e.g. those allowed to introduce new agents to the system) are permitted to make hierarchy updates.

3. **Agent Table Update:** Once the previous two steps have been performed, the registration server updates the agent table to reflect the new services offered by the agent.

3.3 Type Server

The type server can be accessed either directly (this is the mode agents will use to access it) or through a graphical user interface via the registration server (which is the mode users will use when registering a new agent). In the latter case, the type server allows the user to browse and/or search the type hierarchy in the same way as in registration server's hierarchy search and browsing capabilities described above.

In the former case, agents may contact the type server with queries of the form "Is type τ_1 a subtype of type τ_2?" Such queries arise when one agent wants to see if it can pipe certain information it has (of type τ_1) to another agent that requires inputs of type τ_2. Answering such a query involves nothing more than a straightforward traversal of the type hierarchy.

3.4 Thesaurus Server

The thesaurus server, which we are building on top of a commercial thesaurus system (the ThesDB Thesaurus Engine from Wintertree Software) supports only one request type. This request type provides a word as input, and requests all synonyms as output. The thesaurus server, in addition to providing synonyms as output, "marks" those synonymous that appear in one of the two hierarchies (verb, noun-term).

The thesaurus server can be accessed directly or through the registration server – in the latter case, a graphical user interface is available for human users.

3.5 Synchronization Component

One problem with the use of IMPACT servers is that they may become a performance bottleneck. In order to avoid this, we allow multiple, mirrored copies of an IMPACT server to be deployed at different network sites. This solves the bottleneck problem, but raises the problem of consistency across the mirrored servers. To ensure that all servers are accessing the same data, we have introduced a synchronization module. Users and agents do not access the synchronization module. Everytime one copy of data structures maintained by an IMPACT server is updated, these updates

are time-stamped and propagated to all the other servers. Each server incorporates the updates according to the time-stamps. If a server performs a local update before it should have incorporated a remote update, then a rollback is performed as in classical databases [8].

Notice that the data structures of the IMPACT server are only updated when a new agent (or a new service) is added to an existing agent's service repertoire. As the use of existing agents and interactions between existing agents is typically much more frequent than such new agent/service introductions, this is not expected to place much burden on the system.

4 Implementation and Experiments

At this point, we have implemented the algorithms underlying all parts of the **IMPACT** architecture shown in Figure 1 except for the synchronization component, though we do remark that some graphical interfaces are still under construction. Wherever possible, we have tried to use commercially available software – for example, the thesaurus we use in **IMPACT** is ThesDB, and the agent table is implemented using Oracle. The **IMPACT** servers themselves are distributed – for example, the agent table implemented in Oracle does not have to be on the same machine as the registration agent – the administrator of an **IMPACT** server can choose to put them on different machines if s/he wants.

In this paper, we merely report on experiments associated with our nearest neighbor and range query algorithms. We evaluated the performance of these algorithms as the number of nearest neighbors requested increased in number, and as the range of the range query increased in size. In all cases, we used a NASA hierarchy consisting of 17,445 words (solely for experimental purposes, the same hierarchy was used as both a verb and a noun hierarchy although the **IMPACT** prototype uses different hierarchies). Weights on all edges in the hierarchies were assumed to be 1 and the composite distance function was taken to be sum.

Figure 4 shows the performance of the k nearest neighbor algorithm as k is increased from 1 to 20. For any given k we consider 100 queries, generated randomly. Figure 4(a) shows the average time taken for these 100 queries. Notice that in some cases, even though k neighbors may be requested, we may only get back $k' < k$ answers. Figure 4(b) shows the average time per retrieved answer. As the reader will notice, the average time varied between 0 and 1 second, and rose more or less linearly as k increased. However, when considering the average time per retrieved answer (Figure 4b) we notice that the time taken is more of less constant, fluctuating near 0.2 seconds.

Figure 5 shows the performance of the range query algorithm. Again, we ran 100 queries, and increased the range from 1 to 20 units. Figure 5(a) shows the average time taken for these 100 queries, while Figure 5(b) shows the average time per retrieved answer. The average time per range query stays more or less constant at 1.6 seconds. This number is higher than in the case of k nearest neighbors, but is easily explained by the observation that the number of retrieved answers within r radial units may be significantly larger than r. The average time taken per retrieved answer for range queries is around 0.5 seconds.

5 Related Work

There has been substantial work on "matchmaking" in which agents advertise their services, and matchmakers match an agent requesting a service with one (or more) that provides it. Two of the best known examples of this class of work are given below.

Kuokka and Harada[10] present the SHADE and COINS systems for matchmaking. SHADE uses logical rules to support matchmaking – the logic used is a subset of KIF and is very expressive. In contrast, COINS assumes that a message is a document (represented by a weighted term vector) and retrieves the "most similar" advertised services using the SMART algorithm of Salton[12].

Decker, Sycara, and Williamson[3] present matchmakers that store capability advertisements of different agents. They look for *exact* matches between requested services and retrieved services, and concentrate their efforts on architectures that support load balancing and protection of privacy of different agents.

Our effort differs from the above in the following ways: first, our service descriptions use a very simple, restricted language similar to HTML. By restricting our language, we are able to very clearly articulate what we mean by "similar" matches in terms of nearest neighbor and range queries, as well as provide very efficient algorithms (as demonstrated by our experiments) to implement these operations. Second, the user (or owner of an agent) can expand the underlying ontologies (verb, noun-term hierarches) arbitrarily as needed, and we provide him software tools to do so. To date, we have explicitly encoded (in our language) service descriptions of over forty well known independendently developed programs available on the Web, and this number is increasing on a regular basis. However, we do not address issues such as load balancing and privacy issues addressed by Decker et. al. [3].

With respect to agent architectures, there have been numerous proposals in the literature (e.g., [4, 6, 1]) which have been broadly classified by Genesereth and Ketchpel[5] into four categories: in the first category, each agent has an associated "transducer" that converts all incoming messages and requests into a form that is intelligible to the agent. This is clearly not what happens in **IMPACT** – as noted in [5], the transducer has to anticipate what other agents will send us and translate that – something which is clearly difficult to do. The second approach is based on wrappers which "inject code into a program to allow it to communicate" [5, p.51]. The **IMPACT** architecture provides a language (the service description language) for expressing such wrappers, together with accompanying algorithms. The third approach described in [5] is to completely rewrite the code implementing an agent which is obviously a very expensive alternative. Last but not least, there is the *mediation* approach proposed by Wiederhold[14], whch assumes that all agents will communicate with a mediator which in turn may send messages to other agents. In contrast, our framework allows point to point communication between agents without having to go through a mediator.

There has also been extensive work on collaborative problem solving and negation in multiagent systems (e.g., [2, 7, 9, 11, 13]). As our approach supports point to point inter-agent communication, such negotiations can be built on top of our architecture, and thus this body of work complements

ours.

6 Conclusions

IMPACT is a platform for the creation and deployment of multiagent applications. In this paper, we have described the **IMPACT** architecture and specified how multiple agents may interact with each other by utilizing certain common services, provided by the **IMPACT** service layer, that contains a set of servers. Multiple copies of the **IMPACT** service layer may be replicated and mirrored on the network. Agents may access these servers to register the services they provide, as well as to find agents that provide services they need.

In particular, in this paper, we have proposed an HTML-like syntax through which agents may describe their services. We have also specified how the **IMPACT** server identifies agents that provide services "similar" to a requested service. Similarly is computed through two technical means – through a certain kind of "k-nearest neighbor" search in a metric space, and through certain "range" queries in a metric space. We develop algorithms for such retrievals, and report on how these algorithms are implemented in **IMPACT**, as well as how these algorithms perform on large data sets.

References

[1] W. P. Birmingham, E. H. Durfee, T. Mullen and M. P. Wellman. (1995) *The Distributed Agent Architecture Of The University of Michigan Digital Library (UMDL)*, Spring Symposium Series on Software Agent.

[2] S.E. Conry, K. Kuwabara, V.R. Lesser and R.A. Meyer. (1991) *Multistage Negotiation for Distributed Satisfaction*, IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Distributed Artificial Intelligence, 21(6):1462–1477.

[3] K. Decker, K. Sycara and M. Williamson. (1997) *Middle Agents for the Internet*, Proc. IJCAI-97, Nagoya, Japan, pps 578–583.

[4] L. Gasser and T. Ishida. (1991) *A Dynamic Organizational Architecture For Adaptive Problem Solving*, Proc. of AAAI-91, California, pps 185–190.

[5] M.R. Genesereth and S.P. Ketchpel. (1994) *Software Agents*, CACM, Vol. 37, Nr. 7.

[6] L. Glicoe, R. Staats and M. Huhns. (1995) *A Multi-Agent Environment for Department of Defense Distribution*, IJCAI95 Workshop on Intelligent Systems.

[7] N. R. Jennings. (1995) *Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions*, Artificial Intelligence Journal, 75(2):1–46.

[8] H. Korth and A. Silberschatz. (1986) *Database System Concepts*, McGraw Hill.

[9] S. Kraus. (1997) *Negotiation and Cooperation in Multi-Agent Environments*, Artificial Intelligence journal, Special Issue on Economic Principles of Multi-Agent Systems. 94(1-2):79-98.

[10] D. Kuokka and L. Harada. (1996) *Integrating Information via Matchmaking*, Journal of Intelligent Informations Systems, 6(2/3): 261-279.

[11] J. S. Rosenschein and G. Zlotkin. (1994) *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*, MIT Press, Boston.

[12] G. Salton and M. McGill. (1983) *Introduction to Modern Information Retrieval*, McGraw Hill.

[13] M. Wellman. (1993) *A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems*, Journal of Artificial Intelligence Research, 1:1–23.

[14] G. Wiederhold. (1993) *Intelligent Integration of Information*, Proc. 1993 ACM SIGMOD Conf. on Management of Data, pps 434–437.