Organizing Ad Hoc Agents for Human-Agent Service Matching

W. Pasman
Cactus project, Delft University of Technology
Mekelweg 4, 2628 CD Delft, Netherlands
W.Pasman@ewi.TUDelft.nl

Abstract

Ambient intelligence, distributed agent systems, smart buildings and other emerging architectures leave the user lost between large amounts of autonomous entities or agents. With agents in cars, personal devices and other mobile systems, the set of agents available to the user can change rapidly, adding to the problems. We propose a set of agent relations to organize agents in an ad hoc situation. There are dynamic task-, location- and user relations, creating multiple relation hierarchies in agent space. These hierarchies allow robust, space- and task limited, context-sensitive search through agent space. We show how to use this structure to keep user interfaces near the user, and how it can be used to support the user in finding the agent he needs in a context-sensitive way.

1. Introduction

In future ambient intelligence systems, huge amounts of agents will control services ranging from room lighting, heating, coffee machines, total building climate, city touristic routes up to global travel information. An increasing number of those agents will be non-stationary, such as agents in cars, personal devices, and movable furniture. Agents may be available only temporarily, depending on the user's position, network availability etc. It would be fairly impossible to ask the user to locate these agents manually, or to have him browse through a complex menu hierarchy to select the appropriate agent.

Preferably the user should be able to express his wishes in natural language. Natural language enables the user to accurately describe what he needs, without the need of a priori knowledge. Next, the user's request has to be understood by the system. Natural language understanding is most robust when the context of the request (type of service targeted, location, etc) is highly restricted. Therefore, parsing and understanding the user's command is best done separately for every service, instead of having a single parser/translator that would need to know every possible service in the system.

Further, to simplify selection of the desired agent, it would help if the system would take into account context information about the user and about the tasks the user is performing. For instance, knowing the position of the user, the question "give me some light" would then only refer to the 'light' agents in the particular room the user is in. Knowing the previous user requests could provide a cue about the new question. Only if a question can not be addressed by a recently engaged agent then the system should de-focus and search in a wider circle.

Once an agent matching the user's request has been found, the system should automatically connect the user to the appropriate agent and its command interface. The selected agent may handle the request itself, but it may also translate the user's request into several requests to other agents that all handle a part of the task.

In this paper we present an architecture to organize such an ad-hoc agent world robustly and efficiently, in a way that supports our goal of enabling human-oriented, context dependent service discovery using natural language. First we structure the agent world by relating agents in task sense and location sense, and by representing relations between the user and the agents. This knowledge is embedded in the individual agents, such that the system suits an ad-hoc environment. Next, we present two essential services running on this architecture: a Service Matcher that searches for agents fitting the user's natural language request, and a 'KeepGUIWithUser' service that keeps the user interface windows with the user as the user moves around in the world.

The paper is organized as follows. We start with a discussion of existing techniques for service matching and their shortcomings. Next, we present our architecture. Then we describe how the ServiceMatcher and KeepGUIWithUser services run on top of this architecture. We also describe a number of supporting agents, to give a coherent system overview. Then we

discuss the robustness of our system. Finally we discuss our prototype implementation and give some performance indications.

2. Previous Work

Usually the term service discovery (**SD**) is used for locating agents for a certain service. We start with a discussion of the existing technical solutions for SD and their problems. The technical solutions mostly ignore the user, and SD usually refers to a one-way mechanism from service name to agent. Some attempts have been done to come to a more user friendly SD, mostly by using context information to select relevant services. We look at the area of natural language processing, where using context information is much more mature.

2.1. Service Discovery

Current SD architectures can be classified into two approaches: the service discovery service approach and the service abstraction approach.

The service discovery service (SDS) approach is based on using a 'yellow page' like service agent. An SDS agent/database/repository host can be asked which agents can deliver a certain type of service. Sometimes additional requirements can be set, for instance in Semantic Web approaches (e.g. [1]). The SDS returns a list of agents including the protocol(s) that each agent in the list uses. The user of the list then has to pick an agent and negotiate with it. To negotiate, the user has to find the protocol specifications, construct messages and negotiate with the agent according to the protocol.

The service abstraction approach is slightly different. The user just picks a protocol that suits his needs, and sends a service request to a 'resolver' agent. The resolver figures out which of the available agents can (best) handle the request, and forwards the request to that agent. The idea is that it does not matter which agent fulfils the request as long as all service requirements as set in the request are met.

Straightforward implementation of either approach results in a single, central SDS [2, 3, 4, 5, 1] or resolver [6, 7]. This makes the system highly vulnerable to network failures, and thus not robust for the ad hoc situations we foresee for the future.

Several attempts have been made to distribute the knowledge about available services over the system.

At one extreme, every node can be checked individually until the service is found (e.g., [8]). This approach is very robust for network failures, missing services etc, and it will always find the best matching service. Unfortunately it floods the network with service requests, which is extremely energy inefficient and slow.

At the other extreme, all nodes can keep pointers to all available services. For instance in the ACAN system [9] all network nodes keep track of the nearest agent for each service. Chen and Kotz [10] propose a similar system, but here service names can even change at run time, for instance a camera agent may have the name of the room in its name. In a proposal to adapt FIPA agents for ad hoc situations [11] it is proposed to make the SDS optional for a platform, and to reconfigure a cluster of un-managed agents into an ad hoc cluster. One problem with all these distributed approaches is that when the agent having the service descriptions moves out, many agents may have to be informed about modified or broken services, making the reconfiguration expensive.

Another problem with this approach is that in the massive agent scenario we envision, the number of services would be way too large to keep track of at each node. As Ratsimor et al. put it [12], existing approaches are based on the supermarket model (few services, many-customers) while our future scenario more resembles a bazaar model (many services, many customers).

Other approaches as Jini [13] and the Intentional Naming System [14] allow multiple SDSs to be organized hierarchically, allowing unresolved lookup requests to be passed upwards. Although this allows distribution of the SDSs, these systems are not sufficiently robust for ad hoc situations, because every node in the hierarchy is a single point of failure.

Ratsimor et al.[12] avoid keeping track of all services, by using caching of services in the same 'alliance' of partner agents. They propose a lightweight version of a SDS to be available in every agent. If a service is not available in the alliance, they can use multicasting to other alliances in the vicinity – instead of broadcasting – to avoid overloading the network, but they may need broadcasting depending on the situation. Multicasting can target for the 'most powerful' nodes, by checking the cache size of known nodes. This approach seems promising but they offer no measurements or evaluations. Remaining problems with this approach are that only 'good' neighbouring services are checked, ignoring the purpose and known relations of agents. Furthermore there is no mechanism to limit the search scope of a search and the risk stays of flooding the network with requests.

Crespo and Garcia-Molina [15] improve this system. If a node can not answer the service looked for, they forward the query to the node that is most likely to know the answer, instead of to all neighbours. The route indices do not store the target agent for each service, but instead they indicate the neighbours through which that type of services can be found. Furthermore they introduce a maximum hop count, enabling limits on the search scope. They show that with the appropriate heuristics, the performance can be

improved with one to two orders of magnitude compared to the brute-force 'ask all' approach, and up to 100% compared to random forwarding. Although this approach avoids network flooding, it is not clear how the speed of the search compares to a more parallel search as in Ratsimor et al. Furthermore, this approach requires information to be stored for every <service type, neighbour> pair. We assume a huge number of services available, and it is not clear that the lists of all such pairs will stay acceptably small.
Another problem, specific for the service abstraction approach, is that an increasing number of additional constraints will be required if a very specific service has to be targeted, for instance a light near the user. Searching for agents matching all these constraints will become very expensive and overload the network in massive agent worlds.

2.2. Human Friendly Service Discovery

In spite of the huge number of SD protocols, little attention has been paid to supporting a human user in finding useful services. Current technologies all require the user to use the exact naming and ontology as defined by the designers of that technology [16]. As a result, human users unfamiliar with the ontology at best have to browse lists of services, and even users that know the naming and ontology still have to pick one or a composition of services that best serve their specific task from the list. As the number of services increases, learning all related ontologies and names, and picking from a list of matching services will become impractical to totally infeasible [17]. With the service abstraction approach, it may be hard or impossible to specify the exact target agent, because the resolver talks only in abstract terms. This makes service abstraction unusable in certain situations.
Another approach would be to use context information, such as his location, his task at hand, his plans and goals, his previous tasks, etc, to focus the search space and filter out good candidates.
Most frameworks claiming context-aware service discovery have a very limited view on context and its use. In several systems, the user's location and some environment parameters are made available, and the services have to look for themselves what they consider appropriate in this 'context' (e.g., [18], [19]). Ratsimor et al. [12] suggest to use the user's preferences and constraints for service discovery. However they only use these to control the resources and communications of the service discovery service, and not to select the appropriate services.
Context and its use is a critical and heavily researched aspect in the area of natural language processing and understanding. Natural language seems particularly useful for human friendly SD, because it allows the user to be vague or descriptive, and because it allows to specify both his goals and/or his plans. Natural language has been used in a few cases to aid the user for SD. Balke and Wagner [16] start an SD attempt with a keyword (apparently entered by the user) match. Resulting services are requested to make an offer given the user's hard requirements (in their case, a travel scenario with arrival time and location), and then they pick that offer best matching the user's preferences (the 'soft requirements', e.g. 'business class'). However in a massive agent world, such an approach would drown the network with requests for offers.
Coen et al. [20] have a single room equipped with about hundred agents, each one controlling a device. Every agent may listen to the user after the keyword "computer" is spoken by the user. Each agent has its own grammar, optimized for its specialism. It individually monitors that parts of the user's context relevant to him in order to determine whether it actually will listen or not. Using the context and natural language to find the required service is similar to our approach. One problem with their approach is that it is unclear how the room would react on multiple users having different tasks at the same time, it seems that they would heavily interfere with each other. When several similar devices are close to each other and all hearing the user, the user will have difficulties targeting a single device. If a vague request is posed, he might be overwhelmed with responses. Furthermore, as with the approach of Balke et Wagner, their approach seems not to scale to large areas. Finally, the user has to know where the device is and to go there physically before he can address it. This last problem is larger than it looks at first, for instance many services do not have a natural, human size, visible and/or unique physical counterpart.
The Phoenix parser [21] is a speech parser. It aims at a single speech-based application covering multiple services simultaneously, such as flight planning, hotel booking and car rental. Every service has a 'frame' containing the input fields for the request (e.g., departure time and location for a travel planner). The Phoenix parser can try to fit a user utterance to multiple frames [22]. Thus, frame fitting works as a kind of service selection mechanism. Real life tests showed that the frame based approach is very robust and effective. However, this parser has never been intended for fitting a large number of frames (services).
More comprehensive attempts to understand user utterances within the current context have been done for story understanding. On the one hand there is the plan-goal based approach (e.g., [23]), using logic to explain and relate utterances. Several attempts have been made to extend such plan-goal based approach for user interaction (e.g., [24]). However these approaches all have a very heavy, central interpreting core where all knowledge is gathered. We do not see how this

approach could fit in a distributed ambient intelligence situation.

On the other hand there is a more fuzzy approach, relying on estimated 'distances' between concepts that were touched on in the dialog (e.g., [25]). For explaining stories, this approach seems just as effective as the plan-goal based approach. Additionally, the concept knowledge graphs and marker passing mechanisms of Norvig [25] are well suited to adopt for SD in a distributed ad hoc agent architecture.

3. Architecture

The remainder of this paper describes our approach. We introduce an ontology for agent relations to organize the agents. The agents themselves are responsible for knowing and communicating their relations.

Next, we present two crucial services for ad hoc environments: service matching and keeping GUIs with the user. The service matching (**SM**) mechanism is central to our system. It helps the user in locating the service he needs. This mechanism matches a user's request in natural language with the available agents/services, using the user's context to restrict the potentially matching services.

We end with some decisions and findings in our prototype implementation.

We have only limited space here to describe our architecture. For the fine details we refer the reader to the technical documentation of our prototype [26].

3.1. Ontology for Agent Relations

This section presents a simple but effective method to organize an ad hoc agent system. To structure an ad hoc agent environment, every agent is supposed to (or is configured to) know about its context. To communicate this knowledge, all agents speak our "RelatedAgent" ontology. Fig. 1 shows the ontology (in the style of Protégé [27]; indentation means sub classification, and words preceded by ':' are slots). Effectively, this overlays the agent world with a location tree, a task graph and user-related structures. Because every agent knows its place, this builds a robust, distributed system suiting an ad hoc agent world.

Every agent is assigned an AreaSize, roughly indicating the size of the area the agent represents in the real world (meters). This makes area-limited searches possible, for instance to find nearby displays or restaurants. This does not mean that all agents are directly coupled to a physical object, it can also indicate that the agent has control over, or knowledge about that area. Some agents, such as a universal currency converter, might claim a (near-)infinite area size.

```
AgentAction
  addRelation    :relation  aRelatedAgent
  removeRelation :relation  aRelatedAgent
  setAreaSize    :radius    Float
AreaSize         :radius    Float
aRelatedAgent    :agentname AID
                 :tightness Float
  TaskRelated
                 SimilarTask
                 StepOf
                 HasStep
  LocationRelated
                 In
                 Contains
  UserRelated
                 NearbyAgent
                 UserLooksAt
                 RecentlyInteracted
                 CommonTask
                 CarriedByUser
```

Figure 1. RelatedAgent ontology.

The agent relations are all subclasses of aRelatedAgent. As such they all inherit an agent that is related to, and a tightness (a float between 0 and 1). The tightness is defined different for the various relations, but it always attempts to approximate the relevance of this relation in the search for an agent matching the user's natural language query. See the discussion of the service matcher agent for more details on this.

Every agent can claim as many relations as it judges appropriate. What is important is that the agent makes itself related properly, so that it can be discovered and checked by the service matcher when the user needs it. Figure 2 shows a typical message to request an agent to add a relation.

```
(addRelation :relation (StepOf :agentname (agent-
  identifier :name anotheragentname) :tightness 0.7))
```

Figure 2. Example request to an agent, to add a StepOf relation.

The LocationRelations locate agents in the real world. Agents know about larger areas they are contained in (the 'In' relation), and smaller areas that they contain (the 'Contains' relation). Areas can have any shape that is appropriate for the layout and typical use of the space that is modeled. For In- and Contains-relations, the default tightness of a relation is inversely proportional to the distance between the centers of the referred areas.

The task graph relates agents with similar tasks, subtasks (HasStep) or supertasks (StepOf). Here the tightness refers to the probability that a user will do that related task next, assuming he last time did the task related to the agent claiming this relation.

Fig. 3 shows how an agent might be queried for relations, using FIPA-SL [28]. In FIPA, this message

string is embedded in a larger ACLMessage structure before it can be send to an agent.

```
(all ?x (= ?x (LocationRelated :agentname ?y
                tightness ?z)))
```

Figure 3. Example query asking LocationRelated relations.

UserRelated relations form a link between the user and the other agents, pointing to the nearest agent in his environment, the agents representing devices he carries, agents he commonly uses (agenda, word processing agent, etc), and so on.

Most of the relations are static, only a small part changes over time. For instance a bunch of agents coupled to a driving car could all link with a static 'In' relation to the car agent, and only the car agent would have to keep its 'In' relation up to date with the changing environment. Services related to mobile devices (e.g. non-stationary displays, newspaper seller agents, etc.) can link themselves into the relation hierarchy on the fly according to their latest situation (position, task, etc), by requesting relevant agents to update their relations.

Proper configuration of the relations is essential to make our system work. Although this configuration is handwork in our prototype system, we think a significant part of this configuration process can be automated.

3.2. Important Services

This section shows how a few services that are essential for the user's experience, use and handling of the agent world can effectively use our agent relations.

3.2.1. Service Matcher. One important service is the SM agent, that we named ServiceMatcher. For SM, the user describes his needs in natural language. The ServiceMatcher takes this request and negotiates with agents whether they understand the query and can help the user. Once a matching agent has been found, that agent is authorized to handle the request, and may set up a user interface as it thinks appropriate. Users can launch as many ServiceMatcher agents as needed, to do for instance parallel searches. They can be run at any place they like, for instance on their PDA or on a server in the wired network.

For placing the request in the proper context, we were inspired by the ongoing research on story understanding, concept knowledge graphs and marker passing technology [25]. The concepts translate to agents, the knowledge graph was translated to an ad hoc related-agent structure, and the marker passing mechanism translated to a context sensitive search algorithm for SM. This enables us to use numerous context aspects, such as user location and viewing direction, related tasks, recent interactions, etc. to be incorporated in the search for the service the user needs.

The ServiceMatcher accepts an 'AttemptHandling' request with a natural language query (typically typed by the user), coming from a user interface (e.g., a text prompt or a speech to text converter). To attempt handling the request, part of agent space is searched, as spanned by the RelatedAgent structures. This is done by starting with a small scope of potentially matching agents, and gradually extending this scope if the agents within the scope show incapable to handle the user's request.

The search scope initially is set to the PersonalAgent (see below). Then, it is checked whether any of the agents in the scope understands the user's request. If not, the ServiceMatcher 'zooms out' the scope, by asking agents already within the scope about their relations. The scope is extended until an understanding agent is found or the search limit is reached.

Every agent entering the search scope is sent an 'AttemptInterpretation' message in the NaturalLanguageInterface (NLI) ontology (see below). Only NLI agents speak and understand these messages. They return an interpretation message, whether the user's query is understood and if it can be handled (is 'executable'). Other agents will return a not-understood message or do not reply at all.

The internal behaviour of the ServiceMatcher is complex, because non- and late-responding agents have to be dealt with, user interfaces may have to be created and brought into the user's attention to ask him to choose in case multiple agents are found, a map has to be maintained of the agents within scope, and all these things can happen asynchronous. Fig. 4 shows the core of the search algorithm in pseudo code.

```
Initialize search scope
Every second, until agent activated or fail
{
  check how many interpretations received
  0: if scope already extended 8 times
     then fail:limit of search space reached
     else extend scope
  1 which is executable:
    activate agent of that interpretation
  multiple executable:
    ask user which one interpretation
             or agent he wants
    activate agent of that interpretation
  multiple understood, but none executable:
    fail: user is probably asking
             something impossible
}
```

Figure 4. Pseudo code sketching the core of the ServiceMatcher.

Intuitively, to extend the scope we want to 'pressurize' the network from the entry points and check neighbouring agents when their pressure rises above

some threshold. To implement scope extension in a computationally simple way, all agents in the current scope are asked about their RelatedAgents. For every agent that is reported to be related with an agent in the current scope but not yet in the search scope, the 'activity level' is increased by an amount determined by the tightness of the relation. Once the activity level of an agent rises above 1 the agent is incorporated into the current scope.

3.2.2. KeepGUIWithUser. Hopping GUIs between displays to keep them with the user is another aspect that we consider essential for smart environments. The idea is not new (e.g., [29], [30]). However, most current agent architectures are not rich enough to support such a service, and need a lot of extra programming and the addition of a separate infrastructure holding location information of displays. This section shows that our architecture is powerful enough.

A GUI can be automatically kept on a display as close as possible to a user as he moves around. A GUI can do everything necessary itself. First, if it subscribes to the UserLocation tracker agent, the UserLocation agent will send a message whenever a new agent is now closest to the user or being looked at by the user. The agent then can use the LocationRelated relations to find a display within a few meters distance from the user and decide if it is time to move to another display. The LocationRelation hierarchy avoids flooding the network with location requests, and also avoids the need of a central organization or maps locating the displays. Second, the agent can use ping messages to detect if the GUI window dies. In such a case the agent can launch a new window or take other actions.

When the user has many open GUIs, having each GUI doing those steps would cause lots of duplicate work and messaging in all GUIs whenever the user moves. To make life easy for GUIs and to avoid duplication of work and code, GUIs can subscribe to a KeepGUIWithUser agent that will take over GUI movement and maintenance.

3.3. Supporting Agents

This section describes a number of supporting agents in our system. Their functionality is not new, but the organization and use of them is a bit unusual.

Typically these services would be run on the backbone, so that the user can use them and keep in contact with the agent world even if he forgets his PDA or his battery goes flat. For many agents, it could be transparent where an agent actually resides, and the system might even be able to relaunch or relocate agents from crashed machines.

3.3.1. UserLocation. The UserLocation agent keeps track of the agent that the user is closest to, and the agent the user looks at. Other agents can subscribe to get informed if the user's position and/or gaze direction changes.

3.3.2. UserHistory. The UserHistory agent maintains links to agents that the user interacted with recently. If an agent is not used for a long time or if other agents are used, it becomes less likely that the user still wants to contact that agent (using history information only). Therefore, the tightness of a link with an agent halves every 15 minutes and every time the user interacts with another agent. To maintain the links, the UserHistory agent relies on user-agent interaction messages. In our prototype system the ServiceMatcher agent creates those messages.

3.3.3. PersonalAgent. The job of the PersonalAgent is to keep a living connection between the agent world and the user. If that connection breaks down it tries to re-establish it, and it starts up a new connection with the ServiceMatcher if necessary.

The PersonalAgent is also the start point for service matching actions. Therefore it maintains RelatedAgent links with the user's often used and important agents, such as the email, travel planning, UserLocation and UserHistory agents.

3.4. Interface Agents

Besides the RelatedAgent ontology, most agents speak only agent-specific ontologies. We coin the term 'core agent' for such agents. For instance a light core agent will understand some on/off messages and requests for the maximum power, if stated properly in the light-ontology and properly formatted as a message.

Ontologies are not suited for communicating with ordinary human users. Therefore, we introduce interface agents, that can present a user-friendly user interface (e.g., a properly designed GUI) and translate between the user interface events and the messages to and from the core agent. An interface agent is designed for a specific core agent. Multiple user interface agents can exist for a single core agent, for instance for different interface modalities.

This specialization of interface agents for a core service enables the optimization of the UI for that specific task. The separation into a core agent and interface agents is good for lightweight, infrequently used services such as light switches: the heavy user interface can be run on a dedicated user interface processor, while the service itself (the actual switching of the light) can run on a small CPU near the light. Furthermore, this separation promotes interface-independent information storage in the core agent,

which helps to make modality switching more consistent and understandable for the user.

We discuss two important types of interface agents that we also implemented in our prototype system: NaturalLanguageInterface agents and MobileGUI agents.

3.4.1. MobileGUIs. A Mobile Graphical User Interface (MobileGUI) is a normal Graphical User Interface (using the standard WIMP techniques), but it has the ability to move between the displays with preservation of its state (open menus, scrollbar location, selected items, etc). MobileGUIs can be requested to move, both by an agent or by the KeepGUIWithUser service we discussed.

Our MobileGUIs are depending only on Java, making the MobileGUI independent of the underlying window system. Also, by binding part of the software essential for the interfacing to the GUI we avoid intensive GUI messaging (such as mouse moves) between the application and the window system, making the GUI more responsive. In this respect our work is comparable to [31].

3.4.2. NaturalLanguageInterfaces. A NaturalLanguageInterface (NLI) agent is a user interface agent that translates between natural language requests and messages for a core agent. As with all interface agents in our system, an NLI agent is optimized for one core agent, and there are many NLI agents in the system.

Communication with an NLI agent goes in two steps. Step (1) is an AttemptInterpretation request, which asks the NLI agent to interpret a natural language string. The NLI agent will reply with an interpretation: a message whether it (a) understands the query (confidence value between 0 and 1) (b) if it can execute the request (again a confidence value), and (c) how it would execute the request. Step (2) is triggered by an Execute message, after which execution of the proposed interpretation takes place (of course only if the agent thought it could handle the request). If execution is requested, the NLI agent usually will start up some interface with the user to get further details, and then communicates with the core agents to fulfil the user's request.

Step (1) of an NLI agent is well suited for slot filling. Basically this technique does phrase spotting in the sentence to find the data it needs, ignoring words in the sentence it does not understand. Slot filling is a common technique, and is robust for natural conversation which doesn't stick to grammar and spelling rules [32]. Good slot filling parsers are available [21].

Slot filling assumes and exploits the context (the service of its core agent). Only the words and conversion rules that are specific in its context have to be set up. This property is inherited by the NLI agent. Basic grammar rules and vocabulary are supposed to be available in an NLI-agent template, and the programmer of an NLI agent only has to refine the grammar for the context.

Our NLI agent extends the availability of context knowledge of frame based parsers in two ways. First, the NLI agent communicates also the 'executability' of the request, enabling the system to choose those core agents that can not only understand but also handle the request. Second, other context properties such as the location of the core agent can also be used, for instance a NLI agent in China would expect Chinese, in England English language. Our architecture integrates these aspects with the RelatedAgent hierarchy.

The NLI agent has a larger vocabulary than traditional keyword supporting SDSs such as Jini or UDDI, because it has to recognise a wide range of pseudonyms, stop words and language about the context relevant to the service. For instance, an NLI agent for a spotlight lighting a painting would also know basic painting vocabulary, so that it can recognise requests like "light the painting".

The two-step protocol enables higher level processes to decide whether the interpretation is good enough, before an NLI agent is authorized to help the user. Because the way the request will be executed is made available, the user does not have to re-state information he already mentioned, if this service would be selected. It also avoids problems in situations where the meaning of the request could change over time, for instance because of context changes.

All these properties enables the NLI to stay lean, compact, fast, accurate and context-sensitive.

An NLI agent can have various even more specialized input and output interfaces, for instance its input side could be coupled to speech parsers, a standard text prompt or handwriting recognition.

3.5. Robustness

Our mechanisms were all designed for robustness in ad hoc situations. For instance, if one or more agents fall out of the system or move to another location, the UserLocation agent will give other, now available nearby agents from which nearby similar services or displays may be found. Non-responsive and unreachable agents are timed out, and the services (service matcher, mobileGUIs, ...) move on using the agents that responded. The only effect of failing agents is that the system will look further for an alternative agent fitting the request.

All agent-internal errors are caught within the agents and translated into detailed failure messages to the calling agent, so agents can survive even dramatic internal failures properly. A problem with this is that the calling agent may be unsure about the status of his

request: was it completed, still in progress or cancelled somewhere halfway? We defined some general failure classes indicating those cases, so that the calling agent can determine the case without digging through stack dumps and the like. The calling agent then might try an alternative approach to help the user, or inform the user about the failure. The system as a whole just continues in spite of some malfunctioning agents/services, and if the agent properly adjusts its executability estimations the user might find another, more successful alternative the next time he needs a comparable service.

Because each agent has a very limited task scope, this should result in a robust system resembling frame-based systems [21]. Furthermore, our approach enables parallelization of frame-based parsing, which can speed up the process.

4. Prototype Implementation

Our architecture is functioning as a test bed within the encompassing Cactus project [33], which is researching technological and usability aspects of human-computer and computer-network interaction with personalized, intelligent and context-aware wearable devices in future ambient intelligent environments. The primary goal of our prototype is to demonstrate that our architecture enables users to accurately find the services they need in such an environment, using natural language requests.

For our prototype we complied with the FIPA agent standards [28]. JADE [34] was used to implement and run FIPA agents. JADE is based on Java [35] and can run on desktop PCs and PDAs. A lightweight version of JADE named LEAP [36] is available for PDAs and mobile phones (although LEAP for mobile phones has incompatible GUI support). All our code is fully compatible with LEAP.

FIPA agents are not really ad hoc because it is not specified how multiple directory facilitators (the FIPA name for SD agents) would synchronise, and it seems that the entire system breaks down if the infrastructure running the SDS fails. To make a truly ad hoc agent system, we ignored the directory facilitator and used our RelatedAgent structures instead. We only use the underlying FIPA message transfer mechanisms and mechanisms working in individual agents.

The slot filling aspect of the NLI agents was not implemented, because the implementation effort would go way beyond our means. Instead we calculate the distances between the words in the user's request and the words in the vocabulary of each NLI agent. The distance to the agent's vocabulary translates into a value for the understanding. The executability value is set to 1 for now. In our current prototype agent world, most vocabularies are smaller than fifty words, and we have an additional 200 words of the basic dictionary.

The mobileGUI functionality is unaffected by this simplification, but of course it will reduce the accuracy of the matches found by the ServiceMatcher. Also, without slot filling, the agent finally selected by the ServiceMatcher will start with a default interface, ignoring what the user requested to find it. But we expect that our main goal, showing the effectivity of the architecture and ServiceMatcher, will still be reached.

For the same reasons, we use a simple text input window and keyboard to get the user's request. Handwriting and speech recognition could be added in a simple way.

First test results on a simple agent world of 50 agents showed to have acceptable speed. Typically the ServiceMatcher responds in a few seconds. A few parameter settings are the result of simple tests we did with this prototype implementation. We expect that these parameters have to be fine-tuned as we gain experience with larger systems. For example, service matching might slow down if our current limit of eight search scope extensions would show too restricted. Assuming that the average connectivity (number of relations) of the agents in our test system is representative for a large agent world, the total size of the agent space should not have a big impact: we never search the full agent space because we limit our search to eight hops from the current context.

5. Conclusions, Future Work

We presented a mechanism to organize an ad hoc agent system, and showed how it can be used to provide services that are essential in a smart environment. We discussed a few 'crisis' situations in an ad-hoc environment, and explained how our system robustly handles the situation. We avoid user overload by having the ServiceMatcher agent negotiate with the services, until it is clear which agent can really help the user.

One remaining problem of our fault-tolerant agents is that the user may not understand why the service he expects is not available or acting unusual, because everything looks fine at the surface. Only in exceptional cases, the user may get very detailed but hard to read errors if a failure deep in the system causes a chain of errors that no agent can handle. Error handling in smart environments is complicated and needs more research.

Currently we are designing a large experiment involving human users doing real-world tasks with our system. A multi-location test environment with over 500 service agents running on 12 computers has been created. This experiment should answer questions about how well the user can find a service, and how well the service found matches his requirements.

6. Acknowledgments

This research is supported by the Towards Freeband Communication Impulse of the technology programme of the Ministry of Economic Affairs in The Netherlands.

7. References

[1] L. Li, and I. Horrocks, "A Software Framework for Matchmaking Based on Semantic Web Technology", Proc. 12th Int World Wide Web Conference, May 20-24, 2003, Budapest, Hungary.

[2] T. Howes, "LDAP: Use as directed", 1999, http://www.networkmagazine.com/article/DCM20000502S0039.

[3] Globus, "The Globus Project", www.globus.org, 2003.

[4] FIPA, "FIPA Agent Management Specification", http://www.fipa.org/specs/fipa00023, 2002.

[5] "Universal Description, Discovery and Integration of Web Services", www.uddi.org.

[6] OAA, "The Open Agent Architecture". SRI International, Menlo Park, CA, 2001. http://www.ai.sri.com/~oaa.

[7] S. Schubiger-Banz, S. Maffioletti, B. Hirsbrunner, and A. Tafat-Bouzid, "Providing service in a changing ubiquitous computing environment", Proc. of the Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality (HUC 2000), September 2000.

[8] M. Ripeanu, "Peer-to-Peer Architecture Case Study: Gnutella Network", 2001, http://www.eecs.harvard.edu/ ~jonathan/p2p.

[9] M. Khedr, and A. Karmouch, "ACAN-Ad hoc Context Aware Network", Proc. Canadian Conference On Electrical & Computer Engineering (CCECE'02), Winnipeg, Canada, May 12-15, 2002.

[10] G. Chen, and D. Kotz, "Context-aware Resource Discovery". Proc. First IEEE Int Conf on Pervasive Computing and Communications, March 2003, pp. 243-252.

[11] J. Lawrence, "LEAP into Ad-Hoc Networks", Proc. Workshop on Ubiquitous Agents on embedded, wearable, and mobile devices, University of Bologna, Italy, July 16, 2002.

[12] O. Ratsimor, D. Chakraborty, S. Tolia, D. Kushraj, A. Kunjithapatham, G. Gupta, A. Joshi, and T. Finin, "Allia: Alliance-based Service Discovery for Ad-Hoc Environments", ACM Mobile Commerce Workshop, September 2002.

[13] K. Arnold, B. O'sullivan, R. W. Scheifler, J. Waldo, and A. Wolrath, "The JINI specification", Addison-Wesley, Reading, MA, 1999.

[14] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system", Operating Systems Review, 1999, 34 (5), pp. 186-201.

[15] A. Crespo, and H. Garcia-Molina, "Routing Indices for Peer-to-Peer Systems", Tech Report, Stanford University, Computer Science Department, March 2002, http://dbpubs.stanford.edu:8090/pub/2001-48.

[16] W. Balke, and M. Wagner, "Towards Personalized Selection of Web Services", Proc. 12th Int. World Wide Web Conference (WWW 2003, Budapest, Hungary), 2003.

[17] B. Beute, "Navigating Distributed Services", Doctoral Thesis, Center for Tele-Information, Technical University of Denmark, September 2002. ISBN 87-90288-15-7. ISSN 1601-720X.

[18] D. Salber, A. K. Dey, and G. D. Abowd, "The context toolkit: Aiding the development of context-enabled applications", Proceedings of the CHI'99 Pittsburgh, PA, May 15-20, 1999.

[19] P. P. Maglio, and C. S. Campbell, "Attentive Agents", Comm. ACM, 46 (3) (March), 47-51.

[20] M. Coen, L. Weisman, K. Thomas, and M. Groh, "A Context Sensitive Natural Language Modality for an Intelligent Room.", Proc. 1st International Workshop on Managing Interactions in Smart Environments (MANSE'99), Dublin, Ireland, December 1999, pp. 68-79.

[21] CSLR, "The Phoenix Parser User Manual", http://communicator.colorado.edu/phoenix/Phoenix_Manual.pdf .

[22] P. C. Constantinides, "Scoring techniques for Phoenix Parses", In the Phoenix documentation, http://fife.speech.cs.cmu.edu/Phoenix, 1999.

[23] R. Wilensky, "An AI Approach to NLP". Lecture notes for CS288, Computer Science Sivision, University of California, Berkeley, 2002. http://www.cs.berkeley.edu/~wilensky.

[24] W. Wahlster, "Multimodale Interaktion und Interface Agenten: Trends für Morgen und Übermorgen", Keynote speech, UseWare'02, Darmstadt, June 12, 2002.

[25] P. Norvig, "A unified theory of inference for text understanding", Technical Report CSD-87-339. Berkeley, CA: Computer Science Division, University of California, Berkeley, 1987 ftp://sunsite.berkeley.edu/pub/techreps/CSD-87-339.html.

[26] W. Pasman, "Cactus Testbed", Internal Report, Cactus Project, Delft University of Technology, September 2003, http://graphics.tudelft.nl/~wouter/publications/publ.html.

[27] Protégé, "Welcome to the Protégé Project", 2000, http://protege.stanford.edu.

[28] FIPA, "The Foundation for Intelligent Physical Agents", www.fipa.org, 2003.

[29] T. Richardson, F. Bennett, G. Mapp, G., and A. Hopper, "Teleporting in an X Window System Environment", IEEE Personal Communications, August 1994.

[30] B. Brumitt, J. Krumm, B. Meyers, and S. Shafer, "Ubiquitous Computing & The Role of Geometry", IEEE Personal Communications, 7 (5. October, Special Issue on Smart Spaces and Environments), 2000, pp. 41-43.

[31] J. Bates, D. Halls, and J. Bacon, "Middleware Support for Mobile Multimedia Applications", ICL Systems Journal, November 1997.

[32] E. Kaiser, "Robust Parsing: Tutorial", Internal Report, Center for Spoken Language Understanding, Oregon Graduate Institute of Science and Technology, 1998. http://www.cslu.ogi.edu/people/kaiser/profer.

[33] Freeband Cactus Project, "CACTUS develops the personalised electronic assistant", http://www.freeband.nl/projecten/cactus, 2002. See also http://www.cactus.tudelft.nl.

[34] JADE, "Java Agent Development Framework", http://jade.cselt.it, 2003.

[35] Sun, "Java Technology", www.sun.com/java, 1994.

[36] IST, "Lightweight Extensible Agent Platform", Information Society Technologies (IST) project IST-1999-10211, 1999. http://leap.crm-paris.com.