RAJA - A Resource-Adaptive Java Agent Infrastructure *

Yun Ding, Rainer Malaka
European Media Laboratory
Schloss-Wolfsbrunnenweg 33
69118 Heidelberg, Germany
Yun.Ding@eml.villa-bosch.de
Rainer.Malaka@eml.villa-bosch.de

Christian Kray
DFKI GmbH
Stuhlsatzenhausweg 3
66123 Saarbr,cken Germany
kray@dfki.de

Michael Schillo
Multi-Agent Systems Group
Saarland University/DFKI GmbH
Im Stadtwald, Building 36
66123 Saarbr,cken, Germany
schillo@virtosphere.de

ABSTRACT

This paper presents RAJA, a Resource-Adaptive Java Agent Infrastructure. RAJA is easily accessible to agent developers, since it allows structured programming by using a multi-level architecture to clearly separate domain-specific functionality from resource and adaptation concerns. It is generic, since it is applicable to a wide range of adaptation strategies. These two key features are illustrated by several applications, where the RAJA concept has been successfully applied to solve real world problems. They stem from very different domains (video streaming and spatial reasoning), which demonstrates the wide range of application and the flexibility of the proposed infrastructure.

1. INTRODUCTION

The trend of decentralized computing is a continuous phenomenon in information technology. In the early stages, main-frame computers were replaced by desktop computers, and more recently mobile computers that are still gaining popular are already being confronted by two new paradigms: pervasive [29] and wearable (mobile) computing [13].

The basic idea behind these approaches is that the physical environment is augmented with embedded computing and communication devices. Situated in such an environment, users might interact with hundreds of computers at a time, a number of which they might be carrying around with them. Moreover, they may move around and thus experience a highly dynamic environment. In this scenario, interaction and computation become quite dynamic and complex: software agents move along a global network, compete over and bargain for limited resources, and have to adapt to rapidly changing resource availability (e.g. computing power or network bandwidth). Thus, in order to work properly, and to guarantee a certain level of real-time requirements and quality of service (QoS), applications (or software agents) are required to continuously adapt to the current resource situation. This includes, for example, adjusting resource demands dynamically using multi-fidelity[26], anytime algorithms, or by migrating to more powerful sites if minimum requirements cannot be fulfilled at the current location.

The agent paradigm has emerged as one of the most promising approaches to address the challenges of pervasive and mobile applications in highly dynamic and complex environments. Even though there is no general agreement on a single definition of what an agent is, the following is the one most suitable in our context [30]:

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives.

An agent perceives its (changing) environment and is able to act *without* the intervention of humans or other systems. Especially by exploiting parallelism, distribution and mobility, multi-agent systems (MAS) promise to be more powerful and efficient than conventional programs [2], e.g. by making better use of available resources.

In the context of pervasive and mobile computing, a number of middleware systems (not necessarily agent-based) for resource-adaptive applications have been developed. However, most of them are limited to a small set of specific adaptation strategies. We consider the challenge of designing a new *agent infrastructure*[1] for resource-adaptive applications matching the following requirements: First of all, it should be *generic*, i.e., independent of a particular adaptation strategy and thus be able to support a broad range of adaptation strategies. These strategies can range from applying multi-fidelity, anytime algorithms, exploiting the advantages of distributed systems using load balancing and agent mobility to more elaborate strategies. Secondly, it should support both resource-adaptive and resource-unaware applications. Thirdly, it should be *easily accessible* to agent developers and allow *structured programming* of adaptive systems.

The remainder of this paper is organized as follows. We first describe related work in the fields of agent infrastructures and resource-awareness. Section 3 gives a detailed

*The work is funded by the Klaus Tschira Foundation and the German Federal Department for Education, Science, Research, and Technology (BMBF) in the scope of project EMBASSI under FKZ 01 IL 904 D2.

[1]According to Shehory [27] a (multi-)agent infrastructure describes both the *agent internal architecture* and the *multi-agent organization*, and the dependencies between the two.

AGENTS'01, May 28-June 1, 2001, Montréal, Quebec,Canada.

presentation of our novel agent infrastructure for resource-adaptive systems, RAJA. Then, we give a more detailed description of the underlying implementation and its philosophy (section 4). In section 5, we present three examples of resource-adaptive applications, which were developed using RAJA. In section 6, we compare RAJA with other systems. In the concluding section, we sum up the main characteristics of our concept, and point out future research directions.

2. RELATED WORK

KQML [7] and FIPA specifications [8] are standardization approaches for the *interoperability* of multi-agent systems. Since then a number of agent infrastructures, which provide tools for building agents and enabling them to communicate via the KQML or the FIPA agent communication language have been developed. Examples are the KQML-compliant *Jackal* [4] and *JATLite* [16] as well as the FIPA-compliant *JADE* [3] and *FIPA-OS* [9]). RAJA can be viewed as an extension of them, which additionally supports resource-adaptivity. Actually, the current RAJA infrastructure is implemented on top of *JADE* and *FIPA-OS* (see section 4).

In order to support mobile computing, the FIPA 2000 specification [10] includes an ontology for representing the quality of service of the *message transport service* and two additional system agents: a *Monitor Agent* and a *Control Agent*. The Monitor Agent measures the quality of service of a message transport service, collects information from other measuring sources in a repository, and performs first level analysis of the collected data. The Control Agent (CA) controls a message transport connection and selects a message transport protocol using negotiation with other CAs. Agent-based adaptation is achieved by negotiating message transport requirements such as transport protocols or message representation. Implementing RAJA on top of a FIPA 2000-compliant software would allow it to benefit from these services. However, adaptation supported by RAJA is beyond this "technical level", i.e., physical layer of message transport.

DECAF (Distributed Environment Centered Agent Framework) is a modular platform for the rapid design, development, and execution of intelligent agents [14]. It provides a range of architectural services such as communication, planning, scheduling, execution monitoring, and coordination. The framework supports different types of adaptation including organizational, planning, scheduling, and execution-time adaptation [5]. The current implementation of RAJA supports organizational and execution-time adaptation. The flexibility of our infrastructure, however, allows for its extension to support other types of adaptation (see section 7).

The *Odyssey* architecture consists of a set of extensions to operating systems to support mobile, adaptive information access applications [23, 22]. Its central idea is that the system monitors resource availability, notifies applications of the relevant changes (through upcalls), and enforces resource control. This extended system-level functionality is embedded inside a component called *viceroy*. *Odyssey*'s approach to adaptation is to adjust the fidelity of accessed data to match available resources. *Wardens* are the components, which manage data type-specific information (e.g. available fidelity levels of a specific data type). Viceroy and wardens communicate through method calls and shared data structures. In this library-based approach each resource-adaptive application is incorporated with a warden. When notified (by upcalls), an application chooses a fidelity level from a range of levels offered by the warden.

Agilos is a middleware framework for quality of service adaptation, which is achieved in two levels through the components *adaptor* and *configurator* [17, 18]. At the system-level, each adaptor is responsible for a specific type of resource, e.g. CPU or network bandwidth. It ensures a fair and stable distribution of the available resources among concurrent applications. At the application-level, each configurator only serves one application. It takes knowledge about applications into account and maps adaptation decisions made by the adaptor to application-specific parameter-tuning or reconfiguration choices within the application.

The *TLAM* (Two Level Actor Machine) meta-architecture for open distributed systems is composed of base actors and meta actors [28]. Meta actors are part of the runtime system which provide a set of core services for distributed systems. These are, for example, communication, resource management, remote creation, migration, load balancing, scheduling, synchronization, and replication. A meta actor can access and modify information within a base actor.

While there are many different approaches to resource-adaptation, the term *resource* itself also has many-sided meanings: It is a key concept in the context of bounded optimality [25]. It is not restricted to *physical system resources* such as CPU performance or memory and network capacity, but also covers, for example, *users' resources* like time constraints, working-memory load, emotion, or state of health. In addition, there are *situational/contextual resources* determined e.g. by the users' location and their social environment. The project *Resource-adaptive cognitive process* [15] investigates the adaptation of cognitive processes to limited resources, whereby its focus is on *user-oriented resource-adaptivity*. A user's time constraints and working memory limits are treated as key resources, whose limitations are recognized on the basis of the user's behavior. A system takes them into account and adapts its own behaviour accordingly. We concentrate primarily on system resources. However, the RAJA infrastructure is flexible to also support adaptivity for other kinds of resources (see example in section 5.3).

3. THE RAJA AGENT INFRASTRUCTURE

RAJA is a *multi-level* agent architecture (see figure 1). At the application-level *basis agents* model domain-specific application functionality. Each of them can directly contact any of the other basis agents via message passing. At the system-level *meta agents* are part of the agent infrastructure. They provide system-level services (see section 3.2) to basis agents as well as to other meta agents. Meta agents communicate with each other via message passing just like basis agents. Unlike the *meta actors* in the *TLAM* model they do *not* directly examine and modify the state of the basis agents, which can only be done indirectly through *controllers*. A controller is attached to a resource-adaptive basis agent, while a resource-unaware basis agent does not have a controller. In its role a controller is comparable to an application-specific *configurator* in the *Agilos* middleware. In the following we give a detailed presentation of the RAJA infrastructure components and the interactions among them.

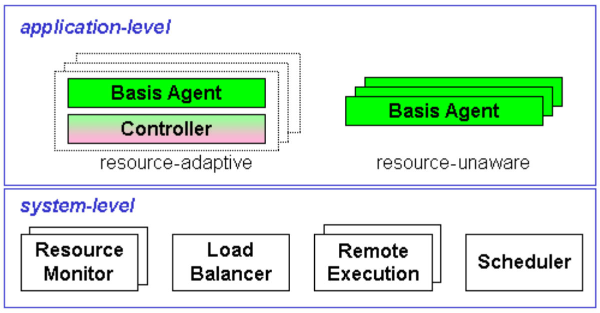

Figure 1: A reflective multi-level architecture

3.1 Basis agents

While a basis agent performs (resource-unaware) *domain-specific computation* to produce certain application functionality, its attached controller performs *reflective computation* [19] *about* the basis agent, more precisely, *about* the domain-specific computation the basis agent performs. Examples of reflective computations are monitoring of performance, (e.g., execution time or result quality), monitoring of resource consumption, and computation of adaptation decision (e.g., which parameter or configuration should the application computation run with next and where should it run). Thus reflective computation can be both merely passive, e.g., the task of monitoring, and active, i.e., it influences or controls the application computation. Due to the separation between basis agents and their controllers (each controller is run in a separate thread) parallel execution of application and reflective computation can be achieved.

In order to make application-tailored adaptation decisions a controller needs to have knowledge about the individual application. For example, video data can be compressed using different compression algorithms and in different qualities. Each controller has knowledge about the dependency between the application parameters. In speech recognition, for example, the parameter *recording quality* of speech varies with the parameter *surrounding noise level*. A controller also needs to understand the relationship between application parameters and resource consumption. This means, for example in the context of speech recognition, knowing that bad recording quality implies more CPU consumption than usual. Through runtime monitoring the dependency and relationship can be more precisely expressed by numerical values. This knowledge, which is represented by numerical *dependency profiles*, forms the decision base for controllers to make a proper adaptation decision (e.g. how to tune a certain application parameter) under the prevailing resource availability. Sophisticated *learning methods* such as performance profiling [31] could also be deployed in the decision-making process.

3.2 Meta agents

It is desirable that a multi-agent system provides a wide range of infrastructure services. Existing specifications such as KQML and FIPA do not address the aspect of *resource-adaptivity* (FIPA 2000 [10] does but only to a limited extent). FIPA specifications consider the interoperability of agents and define *agent management services*, which include the creation and deletion of agents, *"yellow-page" directory services* for agents to register their services, query services offered by other agents, and *routing of messages* between agents. RAJA enhances a FIPA-compliant MAS by additionally addressing *resource management services* offered by a group of meta agents. They are:

- A *Resource Monitor* is replicated for each computing/commication site. It records the availability of local resources, which range from CPU power and network bandwidth to I/O facilities such as displays and printers.
- A *Load Balancer* acquires load information of the basis agents from their controllers and decides where to perform a pending task (see example in section 5.2).
- A *Remote Execution* carries out the decisions of *Load Balancer* by transferring the pending task to a remote basis agent.
- A *Scheduler* determines the activation order of basis agents in order to achieve certain goals (see section 7).

The RAJA infrastructure provides a set of predefined meta agents. Additionally, it provides tools for agent developers to design and implement their own meta agents (see section 4) whose services can be shared among the basis agents.

3.3 Interactions

Interactions between a basis agent and its controller are achieved through *method invocation* (from basis agent to controller) or *upcalls* (from controller to basis agent). For example, a basis agent invokes a method of its controller whenever it has several alternatives to assign an application parameter to continue its domain-specific computation. The controller is asked to make a recommendation according to the actual resource availability. Either the controller can make the decision locally or it must request services of some meta agents.

Interactions between controllers on behalf of their basis agents and meta agents are achieved by *message passing*. Requests for meta agent services are sent to the service-offering meta agents, which in return reply with messages. The interaction can follow different interaction protocols (e.g. FIPA-request or FIPA-query protocol [8]), which are determined by the particular service requested. A controller retrieves answers from the replies, makes some application-specific adaptation decisions based on them, and then returns a recommendation to its basis agent.

An adaptation process can be initiated either by a basis agent *explicitly* (*explicit reflection* [19]), or by its controller, or a meta agent *implicitly* (*implicit reflection* [19]). A basis agent explicitly activates an adaptation, once its method call to its controller returns. An adaptation process will implicitly be invoked by upcalls to the basis agent, once the controller has observed some substantial performance changes of the basis agent or a meta agent has reported some resource changes to the controller.

4. IMPLEMENTATION

The RAJA agent infrastructure provides a set of Java classes, which can be extended to develop resource-adaptive agents [6]. In the following we outline the classes `BasisAgent`, `Controller` and `MetaAgent` for the components basis agent, controller and meta agent respectively. The basis class `BasisAgent` offers a number of methods for agents to

- perform agent management functions with FIPA system agents such as the Agent Management System, the Agent Communication Channel, and the Directory Facilitator.

- communicate via message passing. Convenient methods are available to create new messages and reply messages, and to send and receive them. In RAJA there are two ways to handle incoming messages for a basis agent. They can either be kept in a local message queue waiting for retrieval by the basis agent, which explicitely calls the method `receiveMessage` or `blockingReceiveMessage`, or they can be forwarded by RAJA to some dedicated *message handlers*, which have been specified by the agent developers.

- specify their task structure. The functionality of a basis agent can be divided into a number of sequential or parallel tasks, where each task can be further subdivided into sub-tasks. The task structure can be both linear and non-linear (e.g. a task structure with branches). With RAJA it can be defined at runtime by dynamically adding and removing (sub-)tasks.

- benefit from agent mobility. Questions like "Where am I", "Where is agent A" or "Which are the available sites to which I can move" can be answered using method calls to the RAJA runtime system.

Each meta agent offers a couple of services. The basis class `MetaAgent` provides methods for meta agents to specify their services, which are derived from `MetaAgentAction` (e.g., the name and actor of each service and the interaction protocol it follows will be specified). Further methods exist for registering the associtated tasks, which implement the functionality of these services. Each meta agent is equipped with a *dispatcher* by the RAJA infrastructure, which receives messages addressed to its meta agent, retrieves the requested actions from the message content, activates the actions and eventually sends away the results according to the currently active interaction protocol. Programmers of meta agents are completely relieved from the burden of message handling. They merely write the code which implements the actual action to be undertaken by the meta agent in order to satisfy the requests.

Whenever called by its basis agent, a controller checks if it can return a result locally. If not, it instantiates a corresponding `MetaAgentAction` and starts the interaction with the service-providing meta agent by calling a method provided by the basis class `Controller`. The instantiated `MetaAgentAction` will then be included into a request message and forwarded to the dedicated meta agent by the RAJA infrastructure. As in the case of meta agents, the RAJA infrastructure provides each controller with a *dispatcher*, which receives messages from meta agents and automatically activates the code, which operates on the replies from the meta agents.

To hide communication latency and enable parallelism, a basis agent can specify whether it wants to wait for the completion of an interaction between its controller and some meta agents blockingly. If it does not, the basis agent continues its domain-specific computation while its controller and some meta agents process its request. RAJA methods are available for basis agents to test and wait for the completion of an interaction.

A number of FIPA implementations are available nowadays [11]. To save programming cost we intended to implement our RAJA infrastructure on top of those. After evaluating several of them we chose the *JADE* Framework [3] and *FIPA-OS* [9] as testbed and implemented *RAJA-JADE* and *RAJA-FIPA-OS* respectively. Some methods of *JADE/FIPA-OS* can almost directly be used for basis agents. Others are used as base for the RAJA methods which have a higher abstraction. In general, the RAJA API follows the FIPA notation to ensure that RAJA can be easily ported to other FIPA implementations. In order to provide as much *portability* of the code implemented using RAJA as possible, useful concepts, which are contained in one FIPA implementation (e.g. *FIPA-OS*) but not in the other (e.g. *JADE*), have been simulated in RAJA. However, it seems to be impossible to design a totally unified RAJA API, which hides all the differences of the key concepts of the underlying FIPA implementations. For example, the concept of agent tasks of RAJA must be mapped onto the *multitasking* model of *JADE* and the *multithreading* model of *FIPA-OS* respectively. The difference of these two models is now visible to the programmers through the RAJA API. Moreover, the current RAJA implementations support the *interoperability* of agent platforms which is the objective of the initiative *Agentcities* [1]. Agents developed under *RAJA-JADE* and *RAJA-FIPA-OS* are able to communicate with each other.

5. APPLICATIONS

An agent infrastructure lives with the services it can provide and the applications it has been used to support. We have implemented the meta agents *Resource Monitor*, *Load Balancer* and *Remote Executor* and used them in developing the following resource-adaptive applications.

5.1 Multi-fidelity video streaming

The architecture of a resource-adaptive video streaming application which transfers videos at different levels of fidelity according to the available resources is illustrated in figure 2.

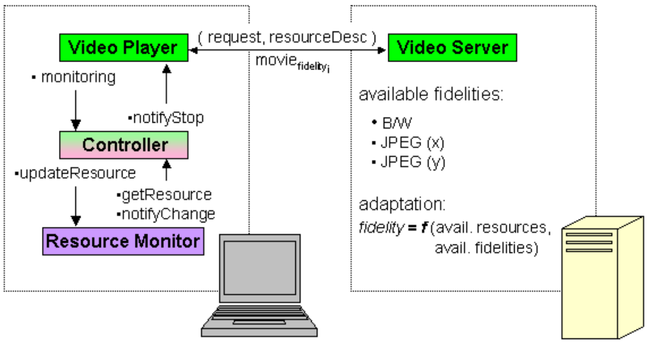

Figure 2: Multi-fidelity video streaming

The video server keeps each movie in several precomputed versions, which differ in their level of fidelity and thus their bandwidth demand. The real availability of bandwidth between the video player and the server can be indirectly measured in *frame loss rate*, which increases with the declining availability of bandwidth. The video player attaches each request for a movie with a resource description (*resourceDesc*), which describes e.g. the local display capability and the estimated available bandwidth (see figure 3). This information is kept by the meta agent, the local *Resource Monitor*, and can be accessed through controllers (*getResource*). The video server sends the movie with the level of fidelity it chose according to the resource description. During the video steaming process the controller of the video player monitors the frame loss rate. Once a certain threshold is exceeded, it estimates the availability of bandwidth and informs the fact both to its basis agent (*notifyStop*) and to the *Resource Monitor* (*updateResource*). When notified the basis agent sends a new request which contains an updated resource description to the video server. The video server is asked to stop the current transmission and send the movie at an adapted level of fidelity from now on. The *Resource Monitor* should be notified by the controller to ensure that the up-to-date resource information can be shared among the basis agents located on the same site.

```
<?xml version='1.0'?> <ResourceDescription>
  <Bandwith>
   <BytesPerSecond reliable='yes'>10000</BytesPerSecond>
  </Bandwith>

  <CPU> <mips>1,34</mips> </CPU>

  <Audio supported='no'> </Audio>

  <Video>
   <fps min='10' max='50' default='30'/>
   <resX>1024</resX>
   <resY>768</resY>
   <numColors>65536</numColors>
  </Video>
</ResourceDescription>
```

Figure 3: Resource description in XML-syntax

In the description above, the video server decides for a fidelity level according to the prevailing resource availability. This is the most natural way, because among all the components it has the most exact knowledge about the available fidelity levels. However, it is also possible, that the controller of the video player itself or with the help of a meta agent, which is specialized in video applications, makes the decision. All the variants can be easily realized with RAJA and additionally, the decision for one of the variants needs not be made at build time but can be made dynamically at runtime. This emphasizes the *flexibility* of our infrastructure design.

We have implemented a video client and a video server agent to stream MPEG movies, whereby the Berkeley MPEG Player *mpeg_play* [21] has been used to display the movies. The controller at the client side intercepts the received MPEG stream and calculates the frame loss rate.

5.2 Transparent load balancing

Dynamic load balancing has often been used to minimize the execution time of single tasks and the overall throughput in parallel and distributed systems. Under RAJA a resource-unaware agent becomes a *load-balanced agent* almost transparently: The default controller provided by the RAJA infrastructure is able to

- monitor the load of its basis agent,
- inform the meta agent *Load Balancer* of the load changes (e.g. from normal to overload and vice versa, or from normal to underload and vice versa),
- find with the help of the *Load Balancer* the most suitable agent, which can handle the pending request in case of local overload, and
- subsequently contact the meta agent *Remote Executor* to transfer the pending request.

This process is totally encapsulated inside the default controller and happens automatically without interfering with the basis agents. However, agent developers have the possibility to deactivate this feature, if load balancing and remote execution are not desirable. The definition of over- and underload thresholds is highly application-dependent. RAJA provides agent developers with methods to explicitly specify initial values for them. During runtime the controller could *learn* the ideal values based on the monitored performance behaviour and *adjust* them dynamically. Figure 4 shows the sample code a load balanced video server agent.

```
public class VideoServer extends BasisAgent {

  public VideoServer ( ) {
   super ();

   // Express the wish of load balancing and
   // set the initial values for under- and overload
   // using the class ResourceAwarenessConfiguration
   ResourceAwarenessConfiguration myConfig =
   new ResourceAwarenessConfiguration ();
   myConfig.setLoadBalancing ( true );
   myConfig.setLoadThreshold ( 1, 5 );
   setResourceAwarenessConfiguration ( myConfig );

   // Initiate an instance of the default controller
   // and attach it to the video server
   myController = new Controller ( this );
   attachController ( myController );
   }

   public void startAgent () {
   // core functionality of the video server
   }
```

Figure 4: A load balanced video server agent

The working principle of a load balanced video streaming using RAJA is outlined in figure 5. The video server is replicated as server A and B. Once the number of waiting requests exceeds the *overload threshold* by a newly incoming request (in figure 5 it is symbolized by the hatched square assigned to the server A), the controller is triggered to perform some load balancing execution. It informs the meta agent *Load Balancer* of the actual local load of video server A (*updateLoadInfo*) and asks for a video server which can satisfy the request instead of server A. Based on the gathered load information and some load balancing algorithms *Load Balancer* finds the most suitable video server, e.g. the underloaded video server B, and answers with its name. Upon getting the answer from *Load Balancer* the controller sends the new request, together with the name of the remote executor, to the meta agent *Remote Execution*. The task of this meta agent is to modify the request, such that the remote executor finds it in the form as if it were directly addressed to it, and to forward the request to the actual remote executor. It is also possible that the new request must be performed by the server A, because the communication cost and the waiting time at a remote server fully nullify the potential saving by a remote execution.

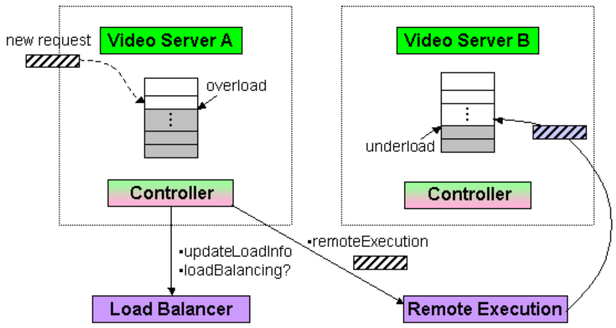

Figure 5: Load balanced video streaming

5.3 A spatial reasoning engine

In the context of the mobile tourist guide Deep Map [20], a component for spatial reasoning SpaCE (Spatial Cognition Engine) was developed, which is currently being ported to RAJA. Deep Map is a multi-agent system, and therefore, SpaCE as part of the system, was designed as an agent. Even more, it is a multi-agent system of it's own, or a so called *holon* [12] consisting of several autonomous subagents, which cooperate in order to solve spatial reasoning problems. This nested or holonic architecture was chosen because there are many (concurrent) processes involved in spatial reasoning, which interact to solve a specific problem. Figure 6 illustrates the logical architecture of SpaCE, which consists of several sub-tasks, for example, *Relations* or *Identification*.

SpaCE makes a good example for a complex agent that might require a great deal of resources, but does also allow for many different resource adaptation strategies to be applied in its sub-tasks. In order to provide a user with easily understandable spatial descriptions and to properly analyze his utterances, extended regions of a detailed world model must be searched, user and context models should be taken into account, and large databases have to be queried for information about the objects found in the model. Furthermore, there are many geometrical computations which must be performed for each object involved.

Since the user expects a reply within a reasonably short time and with a reasonable quality, *resource considerations*

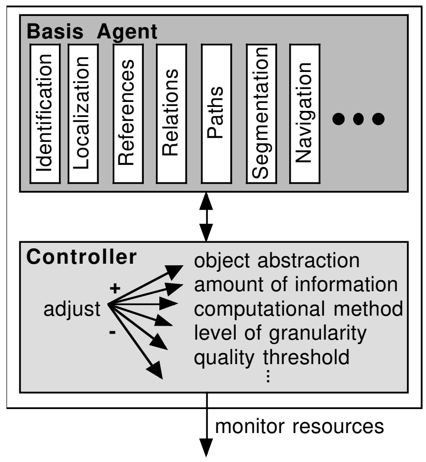

Figure 6: Logical architecture of SpaCE

play an important role in this context. SpaCE was originally implemented on top of a custom FIPA-based middleware written in Java, which does not provide resource handling. Since there is no information on available bandwidth or computational resources, the agent cannot reason about these important factors properly.

Moreover, SpaCE is already very complex even without resource-awareness (e.g. interacting, concurrent processes; multi-factorial object evaluation; etc). Mixing reasoning about resources and adaptation strategies with the core functionality would require a great effort and also make the software too complex to maintain or extend. Using RAJA as middleware, these problems can be avoided and we can take full advantage of the conceptual separation between domain-specific computation and resource reasoning. In leaving the spatial reasoning components of SpaCE mostly unchanged and realizing them as a basis agent, resource adaptation strategies and reasoning can be implemented within the corresponding controller (see figure 6).

Since there are many resources that can be identified in this context, fine-grained adaptation strategies can be formulated. Aside from the overall response time, there are other important resources: the available bandwidth, the computational power available, and the number of agent instances that can be created (e.g. in order to simultaneously evaluate several objects). To adapt to the changing availability of resources, an agent doing spatial reasoning has several options. It can e.g.

- adjust the *level of object abstraction*, which it uses to compute spatial relations. Depending e.g. on the available bandwidth, objects can either be abstracted to a single point, its bounding rectangle, its outline, its bounding box, or it can be analyzed using a full fledged 3D model.
- adapt the *amount of information*, which is taken into account when evaluating different objects. This can range from none to all available information, including e.g. color, function, historical data, etc.
- select different *computational methods*, that are used to establish e.g. spatial relations. This includes purely qualitative calculi, simple region-based reasoning, exhaustive quantitative analysis etc.

- choose a *level of granularity*, which forms the starting point for the analysis. By adjusting this factor, the reasoning can be performed either very roughly, or very precisely, or at several levels inbetween.

- vary the *threshold value for acceptable results*, thereby either speeding up the computation (if the threshold is set to a low value), or yielding high quality results (if it is set to a high value).

Even if one only considers the adaptation strategies in this short list, it is clear that there is a great potential for fine-grained resource adaptive behavior. However, SpaCE is also taking *cognitive* [24], *physical* and *contextual* resources into account. These include, e.g., the (short term) memory load, the attentional focus, and the retention performance of a user, as well as his travel speed, and his current means of transportation. The reasoning that is done on these factors can also benefit from the separation of purely domain-specific computation and resource considerations. By making the adaptation strategies for technical resources and the resources themselves explicit in RAJA, it becomes much easier to map cognitive, physical and contextual resources to technical ones, and to select the appropriate adaptation strategy. In addition, the task of modifying the mapping (e.g. because of new psychological results) is facilitated.

6. DISCUSSION

RAJA fills the gap of supporting *resource-adaptivity* left by existing agent infrastructures promoting agent interoperability. In contrast to the *specifications FIPA 2000*, which are concerned with providing an infrastructure to monitor and control the ever changing resources such as communication connection, RAJA goes a step further by supporting adaptation both at system- and application-level. The proposed *Monitor Agent* is comparable to our meta agent *Resource Monitor*. Compared to *non agent-based middleware* addressing resource-awareness such as *Odyssey* and *Agilos* RAJA distinguishes itself by its greater *flexibility*, which characterizes agent-based approaches. Adaptation decisions can be negotiated between the agents or made corporately at runtime. RAJA differs from *actor- or agent-based approaches* such as *TLAM* or *DECAF* by its concept of *controllers*. The explicit distinction of basis agents and their controllers clearly separates domain-specific computation from reflective computation addressing resource-adaptivity. This makes it possible to convert a legacy *resource-unaware* application into a resource-adaptive application easily by attaching a controller. The separation between controllers and meta agents allows the basis agents to be more *autonomous* (compared to the *TLAM* architecture [28]). Their controllers can use the services of several meta agents simultaneously, choose the most suitable adaptation suggestion or even apply a combination of several of them.

7. CONCLUSIONS AND FUTURE WORK

We presented the RAJA agent infrastructure for resource-adaptive systems. It is a *multi-level reflective architecture*:

- The decoupling of *basis agents* and *meta agents* separates non-functional resource management from application functionality. Algorithms and policies implemented by the system-level meta agents are reusable by different applications. Changes can be implemented without affecting the basis agents.

- The distinction between *basis agents* and their *controllers* allows structured programming and eases the transformation of resource-unaware legacy systems into resource-adaptive systems.

- The separation between *controllers* and *meta agents* isolates application- from system-level resource management and advocates autonomy of agents.

We implemented a prototype of RAJA and have been implementing several resource-adaptive agent systems based on it. We compared our concept with existing specifications, agent infrastructures and resource-aware middleware.

For further work we are interested in providing intelligent *task scheduling* and *task distribution* both at intra- and inter-agent level. At the *intra-agent level* RAJA provides each basis agent with a local *scheduler*, which determines the execution order of the (sub-)tasks contained within the agent. The current RAJA implementations use the default schedulers delivered by *JADE* and *FIPA-OS*, which applies a round-robin non-preemptive scheduling policy and allows tasks to be executed in the order events (e.g. incoming messages) arrive for those tasks respectively. We plan to improve the default schedulers and add the ability to distribute the (sub-)tasks, if possible, to exploit parallelism and assure balanced load in distributed environments. At the *inter-agent level* an incoming request to the MAS should be at first decomposed into sub-requests to several basis agents. This task of decomposition is application-specific and must be handled by agent developers themselves. The RAJA infrastructure could schedule the sub-requests using a meta agent called *Scheduler* and distribute them to basis agents which are possibly located on different sites. Provided with explicitly specified task structures, intelligent local schedulers, and the meta agent *Scheduler*, the RAJA infrastructure supports adaptation both at planning- and scheduling-level [5].

Additionally, we see several points to enhance the RAJA prototype. We will equip the default *controller* with *learning algorithms*, such that adaptation decisions can be made more “adaptively”. For example, when profiting from load balancing the definition of overload threshold could be adjusted according to the performance behaviour. The current meta agent *Load Balancer* uses a simple *sender-initiated* load balancing algorithm, where the overloaded agent activates the balancing process. We intend to support a wider range of load balancing algorithms. Agent developers should be provided with methods to specify one of them for actual use and even to insert their own algorithms. To further ease agent development we plan to offer a *graphical interface* to specify the task structure of basis agents and *visualization tools* to show the local loads of agents, as well as the dynamic distribution of (sub-)tasks, which together achieve certain goals.

Acknowledgments

We would like to thank Dennis Pfisterer for implementing the application “multi-fidelity video streaming”. He has done a great deal of hard work to realize the real-time transport of MPEG data.

8. REFERENCES

[1] http://www.agentcities.org/.

[2] G. Agha and N. Jamali. Concurrent Programming for Distributed Artifical Intelligence. In G. Weiss, editor, *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*, pages 505–537. The MIT Press, 1999.

[3] F. Bellifemine, A. Poggi, and G. Rimassa. JADE - A FIPA-compliant agent framework. In *Proc. The Practical Application of Intelligent Agents and Multi-Agents (PAAM99)*, pages 97–108, 1999. http://sharon.cselt.it/projects/jade/.

[4] R. Cost, T. Finin, Y. Labrou, X. Luan, Y. P. and. I. Soboroff, J. Mayfield, and A. Boughannam. An Agent-based Infrastructure for Enterprise Integration. In *Proc. 1st International Symposium on Agent Systems and Applications (ASA'99)*, 1999.

[5] K. Decker, K. Sycara, and M. Williamson. Intelligent Adaptive Information agents, 1996. Working notes of the AAAI-96 Workshop "Intelligent Adaptive Agents".

[6] Y. Ding. RAJA Programmer's Guide. Technical report, European Media Laboratory GmbH, Germany, 2001.

[7] T. Finin, Y. Labrou, and J. Mayfield. *J. Bradshaw, editor, Software Agents*, chapter KQML as an agent communication language. MIT Press, 1995.

[8] *Foundation for Intelligent Physical Agents. Specifications*, 1998. http://www.fipa.org.

[9] FIPA-OS, 1999. http://www.nortelnetworks.com/fipa-os.

[10] *Foundation for Intelligent Physical Agents. Specifications*, 2000. http://www.fipa.org.

[11] *FIPA Newletter:Inform!*, 2000. http://www.fipa.org.

[12] K. Fischer. Agent-Based Design of Holonic Manufacturing Systems. *Journal of Robotics and Autonomous Systems*, 1999.

[13] N. Gershenfeld. *When Things Start to Think*. Henry Holt & Company, 1999.

[14] J. Graham and K. Decker. Towards a Distributed, Environment-Centered Agent Framework. In *Proc. of International Workshop on Agent Theories, Architectures, and Languages (ATAL99)*, 1999.

[15] A. Jameson. Adapting to the Users Time and Working Memory Limitations: New Directions of Research. In *Adaptivität und Benutzermodellierung in interaktiven Softwaresystemen (ABIS-98)*, 1998. http://www.coli.uni-sb.de/sfb378.

[16] JATLite, 2000. http://java.stanford.edu.

[17] B. Li and K. Nahrstedt. A Control-Based Middleware Framework for Quality of Service Adaptations. *IEEE Journal on Selected Areas in Communications*, June 1999.

[18] B. Li and K. Nahrstedt. QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications. In *Proc. of IFIP International Conference on Distributed Systems Platforms and Open Distributred Processing (Middelware 2000)*, 2000.

[19] P. Maes. Issues in Computational Reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. Elsevier Science Publishers B.V. (North-Holland), 1988.

[20] R. Malaka and A. Zipf. Deep Map challenging IT research in the framework of a tourist information system. In D. B. D. R. Fesenmaier, S. Klein, editor, *Information and communication technologies in tourism 2000*, pages 15–27. Springer-Verlag, Wien, New York, 2000.

[21] The Berkeley MPEG Player. http://bmrc.berkeley.edu/frame/research/mpeg/ mpeg_play.html.

[22] B. Noble. System Support for Mobile, Adaptive Applications. *IEEE Personal Communications*, February 2000.

[23] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. 16th ACM Symposium jon Operating System Principles*, 1997.

[24] R. Parasuraman and D. R. Davies. *Varieties of Attention*. Academic Press, Orlando, Florida, 1984.

[25] S. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of Artifical Intelligence Research*, 1:1–36, 1995.

[26] M. Satyanarayanan and D. Narayanan. Multi-Fidelity Algorithms for Interactive Mobile Applications. In *Proc. 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, 1999.

[27] O. Shehory. Architectural Properties of Multi-Agent Systems. Technical Report CMU-RI-TR-98-28, The Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, 1998.

[28] N. Venkatasubramanian and C. Talcott. Reasoning about Meta Level Activities in Open Distributed Systems. In *Proc. of ACM Principles of Distributed Computing*, 1995.

[29] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[30] M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*, pages 21–35. The MIT Press, 1999.

[31] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, Computer Science Division, University of California at Berkeley, 1993.