ADE: A Framework for Robust Complex Robotic Architectures

James Kramer
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
Email: jkramer3@cse.nd.edu

Matthias Scheutz
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
Email: mscheutz@cse.nd.edu

***Abstract*— Robots that can interact naturally with humans require the integration and coordination of many different components with heavy computational demands. We argue that an architecture framework with facilities for dynamic, reliable, fault-recovering, remotely accessible, distributed computing is needed for the development and operation of applications that support and enhance human activities and capabilities. We describe a robotic architecture development system, called ADE, that is built on top of a multi-agent system in order to provide all of the above features. Specifically, we discuss support for *autonomic computing* in ADE, briefly comparing it to related features of other commonly used robotic systems. We also report our experiences with ADE in the development of an architecture for an intelligent robot assistant and provide experimental results demonstrating the system's utility.**

I. Introduction

Complex robots that interact with humans in typical human environments require the integration of a large set of highly sophisticated techniques from diverse subfields (e.g., various kinds of sensory and perceptual processing, natural language processing and understanding, multiple modes of reasoning and problem solving, etc.). Moreover, these robots will be expected to operate (largely) autonomously in "open computing environments" with little or no intervention from their designers, and thus have to be programmed in a way to maximize their reliability and safety.

While research in robotic systems has addressed many important aspects of human-assistive robotics in the recent past (e.g., [1], [2]; also [3] regarding distributed sensing and control and [4] regarding fault tolerance), no current robotic system provides the necessary *infrastructure features* (e.g., robust distribution techniques, system security, remote maintenance and management, failure detection and recovery, autonomous system reconfiguration, etc.) which we believe are critical in constructing "intelligent robots".

In this paper, we describe a novel architectural framework, ADE, which is well-suited for the development and deployment of complex human assistive robots. Building on work from multi-agent systems (MAS) (e.g., [5], [6]), ADE provides *computing infrastructure plus intelligence*, i.e., support for individual agent architecture development together with the distributed computing infrastructure of a MAS. Specifically, ADE treats architectural components of robotic architectures as autonomous "agents" (in the MAS sense), allowing the implementation of critical features mentioned above. In particular, we focus on the *autonomic computing* [4], [7] aspects of ADE; that is, features that enhance a system's ease of use, availability, and security through autonomous action, including fault-tolerance and system reconfiguration.

II. Complex Robotic Architectures

The need for a robotic framework supporting the systematic parallelization of heterogeneous components of agent architectures over multiple hosts in heterogeneous computing environments is best illustrated by a quote from the authors of the GRACE project [8]. GRACE is one of the most advanced recent robots and relies on many different software packages, four separate processors, and a wireless link to the "outside" world: "One of the more difficult parts of the Challenge for us was determining how to integrate a vast amount of software that had been developed by the participating institutions, mostly on different hardware platforms."

Applications that bring together human abilities and intelligent robots require a wide range of AI techniques (e.g., natural language interaction, vision processing, advanced planning and reasoning, etc.), whose computational demands necessitate the distribution of complex agent architectures over multiple hosts. Furthermore, run-time system modification, reliability (i.e., failure detection and recovery), and automatic system (re)configuration capabilities (aspects of *autonomic computing*), will also be required for safe, robust operation.

The difficulty of integrating diverse software components in a systematic manner is not specific to robotics, but rather a general software engineering problem. There is a need to distinguish between the functional break-down of a robotic architecture into constituent modules (e.g., action planner, plan execution, localization and map-making, feature and object detection and tracking, etc.), their interactions, and the operation of these components in the running system. This division is a recognized issue in agent research; for example, in agent-oriented software engineering, these different aspects of a system have been characterized as belonging to separate "scales of observation" (*micro*, *macro*, and *meso*) [9].

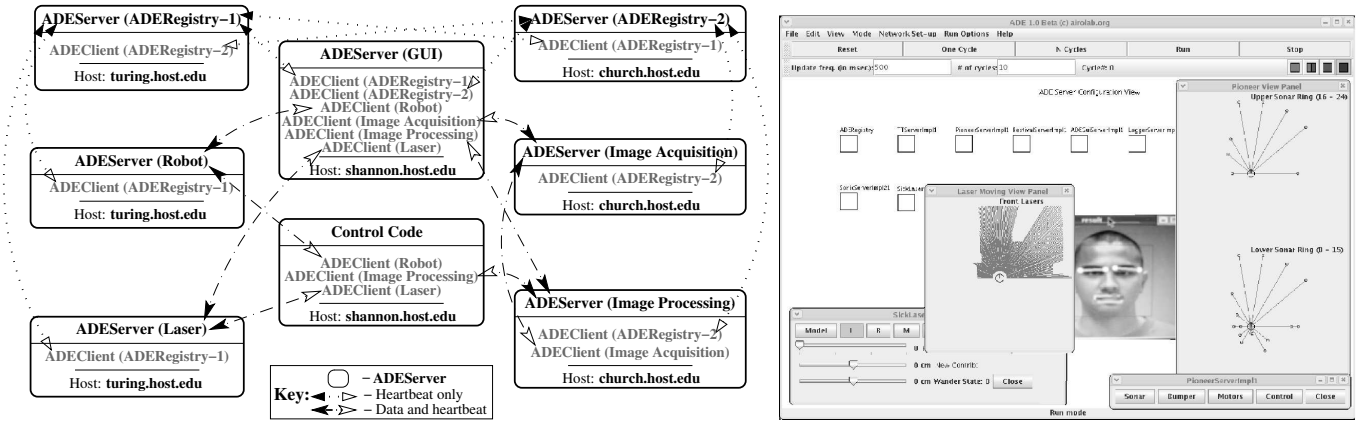

Fig. 1. **Left**: Sample depiction of ADE components. The top, middle, and bottom of each box indicate the component's name, connections, and host, respectively. A description of lines and arrowheads is given in the text. **Right**: Screen capture of the ADE GUI from the application described in Section IV.

Current systems used in robotics tend to focus on the *micro-scale*, providing features such as hardware abstractions, sensory processing routines, simulators, and other tools for the design and implementation of robotic architectures. For instance, Saphira [10], Player/Stage [11], MARIE [12] (with the related RobotFlow and FlowDesigner software), and MissionLab [13] are well-known systems for developing robotic applications that provide design tools and a large selection of implemented functional components that work across many different robot platforms. While all provide mechanisms and methods that support distributed computing, they provide neither the infrastructure nor the tools and techniques normally associated with a MAS (particularly the autonomic computing features listed above).

III. ADE – An Agent Development Environment

ADE is an agent system that combines support for the development of complex individual architectures with the infrastructure of a MAS. It is not an architecture in and of itself; rather, it is a framework based on the theoretical APOC [14], [15] universal agent architecture formalism, implemented in Java, that can be used to express any agent architecture. The left-hand side of Figure 1 depicts an example configuration of components for a basic robotic application, referred to in the following section; the right shows a screen capture of the ADE GUI from the application described in Section IV.

A. An Overview of ADE

The basic component in ADE is the ADEServer, which is comprised of one or more computational processes that serve requests. A set of uniform *interfaces* provides the appropriate abstractions now common in robotic systems of which an ADEServer may implement one, many, or a custom, developer designed interface. Accessing the services provided by an ADEServer is accomplished by obtaining a *reference* to the (possibly remote) ADEServer. The reference forms the *local representation* of an ADEServer, referred to as an ADEClient. ADEServers can provide services to other ADEServers as requested via ADEClients; the result is a *complex* ADEServer, which is a collection of ADEClients and other computational processes. Developers can define new ADEServers to implement their own required functionality (e.g., a novel localization algorithm) and make the services available via ADEClients.

The ADERegistry, a special type of ADEServer, mediates connections among ADEServers and the processes that use their services. In particular, an ADERegistry organizes, tracks, and controls access to ADEServers that register with it, acting in a role similar to a *white-pages service* found in MAS. During the registration process, an ADEServer provides information about itself to the ADERegistry, such as its preferred *host(s)* and *port*, a *name* and *type* identifier, hardware requirements, its *heartbeat* period (details below), the number of connections it can support, and a permission list. This information can also be obtained by an ADERegistry from *startup files* that can contain the entire structure of a particular architecture configuration, as described in Section III-B.

A set of ADE components may contain multiple ADERegistrys. The configuration shown on the left in Figure 1, for instance, makes use of two *global* ADERegistrys (that is, visible to all ADEServers) located on different hosts. When a request for a connection is made, an ADERegistry first scans its list of registered ADEServers for an ADEServer that matches. If none is found, the request is forwarded to the other ADERegistry. An ADERegistry may also be embedded in an ADEServer as a *local* registry, so that ADEServers registered with the local registry are hidden, only visible to the ADEServer containing it (not shown in the figure). A further discussion regarding the ADERegistry is given in Section III-B.

All connected components of an implemented architecture (that is, ADEServers, ADEClients, and ADERegistrys) maintain a communication link during system operation. At a bare minimum, this consists of a periodic *heartbeat* signal indicating that a component is still functioning. An ADEServer sends a heartbeat to the ADERegistry with which it is

registered (indicated by dotted lines in Figure 1, where the solid arrowhead points to the server and the empty arrowhead to the client), while an ADEClient sends a heartbeat to its originating ADEServer (indicated by dot-dash lines, with the same arrowhead conventions). The component receiving the heartbeat periodically checks for a heartbeat signal; if none arrive, the sending component receives an error, while the receiving component times out. An ADERegistry uses this information to determine the status of ADEServers, which in turn determine their accessibility. An ADEServer also uses heartbeat signals to determine the status of its ADEClients, which in turn can determine if an ADEServer's services remain available. Heartbeat and services data have distinct connections between ADEServers and ADEClients, although the dot-dash lines represent both in Figure 1.

A process establishes a connection to an ADEServer by contacting an ADERegistry on a known host and port, sending a user name, password, and request for access. After confirming access permission, the ADERegistry relays the request to an appropriate ADEServer or, if none is found, to another ADERegistry (if one exists). An ADEClient is returned that provides access to the services offered. Once a connection is established, communications are made directly via the ADEClient; the ADERegistry does not play a direct role in their message passing.

B. Autonomic Features of ADE

ADE, like other systems for developing complex robotic applications (e.g., Saphira [10], Player/Stage [11], the Mobile and Autonomous Robotics Integration Environment (MARIE) [12], or MissionLab [13]), provides support for the development of individual agent architectures, pre-defined components to be used in architectures, and the infrastructure for architecture modification, extension, instantiation, and execution. Different from other systems, the ADE infrastructure inherently enhances the "intelligence" of robotic architectures through its use of *autonomic computing* mechanisms. In particular, the ADE infrastructure provides the intertwined features of (1) *system startup*, (2) *failure detection*, (3) *failure recovery*, and (4) *dynamic system reconfiguration*, with no extra effort on the part of the application designer.

System startup in ADE can be performed by an ADERegistry, which receives a list of hosts (potentially) available for application execution and an optional "configuration script" specifying part or all of an application's configuration. Each ADEServer in the script is started after confirming the host on which it is to execute is confirmed as available, allowing the entire architecture–even when distributed across multiple hosts–to be started with a single command. Furthermore, the startup parameters of an ADEServer joining the system later are recorded upon registration, allowing a current configuration to be saved for later duplication. Host availability is confirmed and operating statistics are gathered at startup and throughout application execution, optionally enabling ADE to locate architecture components.

Failure detection relies on the *heartbeat* signals described in Section III. If a component ceases sending a heartbeat signal, it is presumed that either the communication link has been broken or that the component has stopped functioning. There are two cases in which failure detection occurs; the first is when a component (e.g., the "Image Acquisition" server shown in Figure 1) does not send a heartbeat to its ADEClient. Processes using an ADEServer's services invalidate their ADEClient, potentially altering normal function or relaying notice of the failure to other connected components (e.g., the "Image Processing" server relays the error to the "Control Code", which can send a command to the "Robot" server to stop the motors). This provides a reflection mechanism at the architecture level that can be exploited to build rudimentary "self-awareness" into the system (e.g., ADE could put a "fact" such as ImageAcquisition failed at time 14:03 in the database of a deliberative reasoning component). The second type of failure detection occurs when an ADERegistry does not receive a heartbeat from a registered ADEServer. The ADERegistry assumes that the server has ceased operation and disallows new connections.

Failure recovery can be initiated once a failure has been detected. In the first case mentioned above, the component with the now-invalid ADEClient automatically attempts to renew its connection. In the second case, the ADERegistry relies on the startup procedures outlined earlier to attempt to restart the failed ADEServer. Specifically, upon failure detection, the ADERegistry searches its known list of hosts for component relocation. Potential hosts that cannot support the components due to unsupported hardware requirements are filtered out and the remaining possibilities are ordered using the host information supplied in the configuration file and some measure of preference (e.g., the host with the fastest processor, lowest CPU load, or some combination of qualities). The preferred host is then checked for availability. If available, the component is recovered there; if not, the next host in the list is tried until all hosts have been exhausted. Upon successful restarting of an ADEServer, the processes with broken connections will obtain new, valid ADEClients, restoring the system to a fully operational state (e.g., the robot whose "Image Acquisition" server failed would regain its sight, alerting the "Control Code", which would resume sending motor commands).

Since an ADERegistry bears responsibility for restarting components, its failure recovery requires additional mechanisms. The first is ADERegistry replication, where failure causes a "backup registry" to substitute for the original. The second requires mutual registration, as shown in Figure 1. If one ADERegistry fails, the other attempts to restart it. In either case, affected ADEServers will reconnect, as specified above. Even if several subsystems of the architecture fail, components can be relocated and restarted, providing graceful system degradation (if complete recovery is not possible).

Dynamic system reconfiguration in ADE can take various forms, from simple redirection of component connections to a form of controlled, situation dependent failure and recovery.

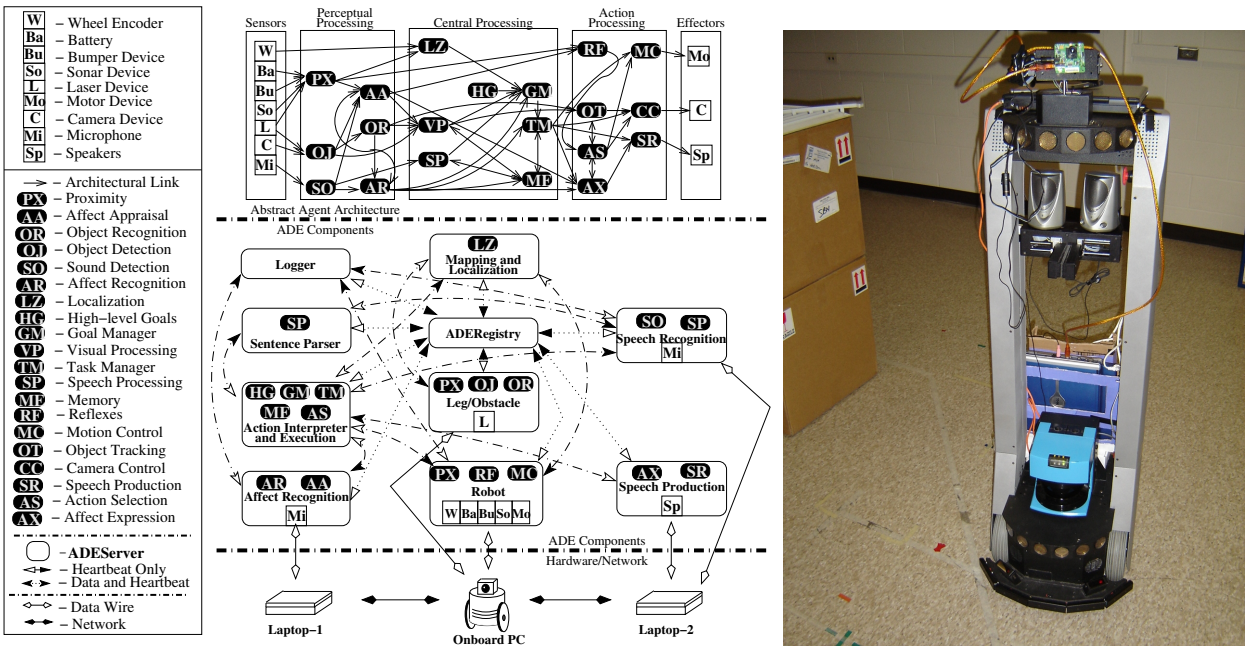

Fig. 2. **Left:** The DIARC Architecture. **Right:** The robot performing the exploration task.

In each case, this allows a "fixed" individual architecture to be transparently changeable. There are at least three reasons for dynamic system reconfiguration–computational load, response time, and reliability. In the simplest case, an `ADEServer` can dynamically change its allowable number of connections, forcing requests for that component to be redirected or fail. An alternative method is to use *component sharing*, which allows a single component to be used by multiple agents, desirable when there are specialized and/or limited resources. Another alternative is to make use of *component substitution*, where one instance of a component is substituted for another. Finally, using the distributed nature of `ADE`, it becomes possible for an `ADERegistry` to intelligently avoid overloading a computational unit if load is high, response time is delayed, or the connection is unsteady, by either redirecting connections to another instance of the component, starting a duplicate component, or stopping the server and restarting it.

Saphira uses plain text-based configuration files (with format similar to Windows .INI files) for robot devices. Due to the lack of distribution infrastructure, components must be started individually and manually on each host, then connected via sockets by making direct calls. Saphira provides callback mechanisms for failure detection due to a broken connection or component crash, but no general recovery mechanisms are available. System reconfiguration must be done manually. While Player/Stage also uses text-based configuration files that contain *device definitions* that must be individually started, the provided naming service can then be used for establishing connections between devices. Player/Stage provides error messages that can be used to identify improper function and component failure, but provides no recovery mechanisms nor dynamic system reconfiguration options beyond manual startup and shutdown procedures. MARIE, which relies on the Adaptive Communication Environment (ACE) [16] for distribution, uses XML files to specify module configuration. Based on the example applications included, *Application Managers* are started with shell scripts that assume modules are already available via the ACE naming service. Although ACE has incorporated the Fault-Tolerant CORBA specifications, it appears that MARIE does not yet make use of them. MissionLab has the most extensive support for startup configuration, supporting both uploading (via FTP) and startup (via `rsh`) of "robot executables" produced by its *CfgEdit* program. However, MissionLab does not provide fault detection, fault recovery, or dynamic system reconfiguration procedures.

IV. Experimental Evaluation

To evaluate the autonomic features of `ADE`, we used an assistive robot that interacts with humans using natural language. Specifically, we rely on the DIARC architecture, which is being used to study the role of affect in joint human-robot tasks [17] and has been demonstrated in a robotic competition [18]. A simplified depiction of the architecture is shown in Figure 2, which is described in more detail below. This is followed by a brief description of the example task, an outline of the experiments performed, and the experimental results.

The "Mapping and Localization", "Leg/Obstacle Detection", and "Robot" servers were directly connected to the robot hardware over serial ports and are responsible for various low-level functionality. Sensory processing duties occur at this level, including obstacle detection and avoidance, leg detection, motor control, and localization and mapping. Speech processing was performed by the "Speech Recognition" (which interfaces with the Sonic speech recogni-

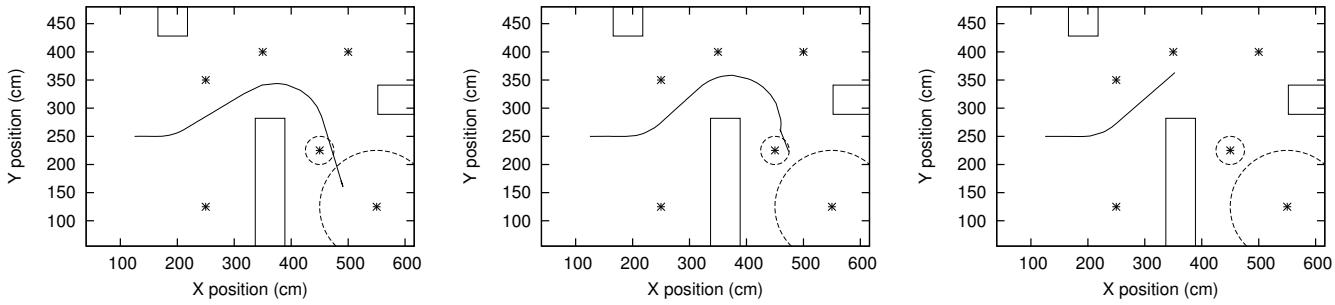

Fig. 3. Three representative experimental task runs; boxes are obstacles, stars are field concentration locations, and circles surrounding a star are valid transmission boundaries. The left-most image shows the best case, where no failure occurs. The middle image shows a failure and subsequent recovery (with no redundant components executing), while the right-most image shows the outcome without recovery. The case with recovery mechanisms and redundant components is not shown, due to its similarity with the middle image.

tion package) and "Speech Production" (which uses Festival software) servers, initially located on "Laptop-2". The "Action Interpreter and Execution", "Sentence Parser", and "Affect Recognition" servers were located on "Laptop-1". High-level planning, action selection, and natural language understanding are performed by the "Action Interpreter and Execution" server, an adapted version of ThoughtTreasure [19], augmented by the "Sentence Parser", which processes text sentence input to return a semantic parse. In case of the "Action Interpreter and Execution" server failure, a chat-bot was added to the "Speech Recognition" server with a special command set (e.g., "stop"), so that the robot could continue to operate, albeit in a restricted manner.

The chosen task is relevant to NASA's envisioned future space explorations with joint robot-human teams [2], taking place in a hypothetical space scenario. A mixed human-robot team on a remote planet needs to determine the best location in the vicinity of the base station for transmitting information to the orbiting space craft. Unfortunately, the electromagnetic field of the planet interferes with the transmitted signal and, moreover, the interference changes over time. The goal of the team is to find an appropriate position as quickly as possible from which the data can be transmitted. The robot is dependent on its human teammate for direction, supplied through natural language commands, while the human is dependent on his robotic teammate for "field strength" readings that cannot be obtained through other means. The task is accomplished when transmission is completed.

Experiments consider simulated *catastrophic hardware failure*; in particular, the effects of failure of an entire computational unit (and, therefore, the software components executing on that unit). The application implementation uses an ActivMedia Peoplebot (shown on the right of Figure 2) with a SICK laser range finder and an on-board 850MHz Pentium III. In addition, it is equipped with two PC laptops with 1.3GHz and 2.0GHz Pentium M processors, each with a microphone and speakers. All three run Linux with a 2.6.x kernel and are connected via an internal wired ethernet; a single wireless interface on the robot enables system access from outside the robot for the purpose of starting and stopping operation. Obstacle detection and avoidance is performed on the on-board computer, while speech recognition and production, action selection, and subject affect recognition are performed on the laptops. We only consider the case of one of the laptops failing; due to the lack of redundant hardware located on the robot "base", its failure would make task completion impossible (e.g., no action can be taken with non-operational motors).

The left side of Figure 2 shows a "3-level" depiction of a possible configuration of the DIARC architecture. The bottom, or "Hardware Level", specifies hosts and their connections. The middle level, referred to as the "ADE Component Level", uses the same conventions for lines and arrowheads as in Section III. Hardware devices used by an `ADEServer` are depicted by a set of labeled squares within a rectangle. The top level is referred to as the "Abstract Agent Architecture Level", where darkened ovals represent architectural components, shown in a data flow progression from sensory input on the left to effector output on the right. Two relations between the bottom and middle levels are shown: (1) `ADEServers` are placed in a vertical column directly above the host on which they execute and (2) connections between hardware devices and the `ADEServers` that use them are indicated by solid lines that cross the separating line. The relation between the middle and top levels consists of a darkened oval that represents a functional architectural component of an agent within an `ADEServer`'s rectangle.

TABLE I

AVERAGE TIME TO COMPLETION (IN SECONDS) OVER TEN RUNS.

Type	Avg. Time (s)	Std. Dev.
No failure	75.47	8.0
With recovery (redundant components)	87.24	14.0
With recovery (no redundant components)	103.19	26.0
No recovery	∞	n/a

To simulate the hardware failure, the network interface on "Laptop-2" is manually shutdown during task execution, such that the "Speech Production" and "Speech Recognition" components are disconnected from the architecture. When the `ADERegistry` does not receive their heartbeats, new instances of the components are started, effectively causing

them to be relocated to "Laptop-1". Experiments record the *time to task completion* to provide a measure of the failure recovery mechanisms, recorded for four separate cases: (1) no failure occurs, providing a best case scenario, (2) failure occurs with recovery mechanisms active and redundant components already executing on "Laptop-1", (3) failure occurs with recovery mechanisms but no redundant components, and (4) failure occurs without recovery, providing a worst case scenario. In each case, the task is performed by a subject who has prior experience and knowledge of the task (so as to keep the probable time to completion for the task approximately fixed); the experimental configuration is kept constant in each case. Time results are shown in Table I, while Figure 3 shows three representative experimental runs.

In the case where these mechanisms are disabled (i.e., similar to the previously mentioned systems), the task cannot be completed because the robot cannot recognize the human speaking, which we mark as taking an infinite amount of time. On average, a successful experimental run with a failure recovery that takes advantage of redundant components took about 12 seconds longer than when there was no failure, while a failure recovery in which redundant components are not available took about 27 seconds longer. (The standard deviation for the recovery experiments is relatively large compared to the "No failure" runs due to the possibility that the robot is facing away from the goal point upon recovery.) The time for failure detection is determined by the period of the heartbeat (that is, the time it takes for an `ADERegistry` or `ADEServer` to miss a component's heartbeat and initiate recovery procedures). The remaining time is due to locating a new computational host and restarting the component.

While the example configuration has only one available computational host with the requisite hardware to support the failed components, it demonstrates `ADE`'s ability to provide robust fault-tolerant system behavior *during task execution*. In addition to the general failure recovery mechanism described, each `ADEServer` is notified when a component to which it is connected fails or comes back on-line; both before and after the connection is re-established, this notification can be used to execute a number of steps that allow it to adjust its behavior internally according to the changed system state (e.g., the loss of the "Speech Recognition" server causes the "Speech Production" server to announce that the robot cannot hear anything.) With this proof-of-concept in hand, we are currently in the process of expanding the set of system information that is used, establishing mechanisms to probe for host information, improve the reasoning criteria, and dynamically tune parameters to optimize application performance.

V. Conclusion

We have argued that future complex robotic applications, in particular, autonomous human assistive robots, will require a distributed computing infrastructure that allows for robust, reliable, and autonomic operation, as well as the ability to dynamically start, restart, relocate, revise, modify, and replace functional components in a running architecture. To meet these requirements, we have implemented the `ADE` architecture framework, which views and implements components of robotic architectures as "agents" (in the sense of MAS agents), thus allowing for the distribution and autonomous operation of architectural components.

References

[1] J. Burke, R. Murphy, E. Rogers, V. Lumelsky, and J. Scholtz, "Final report for the DARPA/NSF interdisciplinary study on human-robot interaction," *IEEE Transactions on Systems, Man and Cybernetics, Part C*, vol. 34, no. 2, pp. 103–112, 2004.

[2] T. Fong, I. Nourbakhsh, and K. Dautenhahn, "A survey of socially interactive robots," *Robotics and Autonomous Systems*, vol. 42, pp. 143–166, 2003.

[3] B. Gerkey, R. Vaughan, K. Støy, A. Howard, G. Sukhatme, and M. Matarić, "Most valuable player: A robot device server for distributed control," in *Proceedings of IROS 2001*, 2001, pp. 1226–1231.

[4] N. Melchior and W. Smart, "A framework for robust mobile robot systems," in *Proceedings of SPIE: Mobile Robots XVII*, vol. 5609, 2004.

[5] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE - a FIPA-compliant agent framework," in *Proceedings of the 4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, 1999, pp. 97–108.

[6] K. Sycara, M. Paolucci, M. V. Velsen, and J. Giampapa, "The RETSINA MAS infrastructure," *Autonomous Agents and Multi-Agent Systems*, vol. 7, no. 1, pp. 29–48, 2003.

[7] D. Bantz, C. Bisdikian, D. Challener, J. Karidis, S. Mastrianni, A. Mohindra, D. Shea, and M. Vanover, "Autonomic personal computing," *IBM Systems Journal*, vol. 42, no. 1, pp. 165–176, 2003.

[8] R. Simmons, D. Goldberg, A. Goode, M. Montemerlo, N. Roy, B. Sellner, C. Urmson, A. Schultz, M. Abramson, W. Adams, A. Atrash, M. Bugajska, M. Coblenz, M. MacMahon, D. Perzanowski, I. Horswill, R. Zubek, D. Kortenkamp, B. Wolfe, T. Milam, and B. Maxwell, "GRACE: an autonomous robot for the AAAI robot challenge," *AI Mag.*, vol. 24, no. 2, pp. 51–72, 2003.

[9] F. Zambonelli and A. Omicini, "Challenges and research directions in agent-oriented software engineering," *Autonomous Agents and Multi-Agent Systems*, vol. 9, no. 3, pp. 253–283, 2004.

[10] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, "The Saphira architecture: A design for autonomy," *JETAI*, vol. 9, no. 1, pp. 215–235, 1997.

[11] B. Gerkey, R. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[12] C. Côté, D. Lètourneau, F. Michaud, J. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran, "Programming mobile robots using RobotFlow and MARIE," in *Proceedings IEEE/RSJ International Conference on Robots and Intelligent Systems*, 2004.

[13] D. MacKenzie, R. Arkin, and J. Cameron, "Multiagent mission specification and execution," *Autonomous Robots*, vol. 4, no. 1, pp. 29–52, 1997.

[14] V. Andronache and M. Scheutz, "Integrating theory and practice: The agent architecture framework APOC and its development environment ADE," in *Autonomous Agents and Multi-Agent Systems*, 2004, pp. 1014–1021.

[15] M. Scheutz, "ADE - steps towards a distributed development and runtime environment for complex robotic agent architectures," *Applied Artificial Intelligence*, vol. 20, no. 4-5, 2006.

[16] D. Schmidt, "The ADAPTIVE communication environment: An object-oriented network programming toolkit for developing communication software," in *12th Annual Sun Users Group Conference*, 1994, pp. 214–225.

[17] M. Scheutz, P. Schermerhorn, and J. Kramer, "The utility of affect expression in natural language interactions in joint human-robot tasks," 2006, ACM Conference on Human-Robot Interaction (HRI2006), forthcoming.

[18] M. Scheutz, V. Andronache, J. Kramer, P. Snowberger, and E. Albert, "Rudy: A robotic waiter with personality," in *Proceedings of AAAI Robot Workshop*. AAAI Press, 2004.

[19] E. T. Mueller, *Natural language processing with ThoughtTreasure*. New York: Signiform, 1998.