CONF-920540--11

DE92 000433

Engineering Physics and Mathematics Division

A System for Simulating Shared Memory in Heterogeneous Distributed-Memory Networks with Specialization for Robotics Applications*

J. P. Jones, A. Bangs, P. L. Butler

Paper submitted to the 1992 IEEE International Conference on Robotics and Automation, May 10–15, 1992, Nice, France

* Research sponsored by the Office of Nuclear Energy, Office of Advanced Reactor Programs, U.S. Department of Energy, under contract No. DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

A System for Simulating Shared Memory in Heterogeneous Distributed-Memory Networks with Specializations for Robotics Applications

Judson P. Jones **Alex L. Bangs**
Center for Engineering Systems Advanced Research
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
P.O. Box 2008, Oak Ridge, TN 37831-6364 USA

Philip L. Butler
Telerobotics Systems Section
Robotics and Process Systems Division
Oak Ridge National Laboratory
P.O. Box 2008, Oak Ridge, TN 37831-6364 USA

Abstract

Hetero Helix is a programming environment which simulates shared memory on a heterogeneous network of distributed-memory computers. The machines in the network may vary with respect to their native operating systems and internal representation of numbers. Hetero Helix presents a simple programming model to developers, and also considers the needs of designers, system integrators, and maintainers. The key software technology underlying Hetero Helix is the use of a "compiler" which analyzes the data structures in shared memory and automatically generates code which translates data representations from the format native to each machine into a common format, and vice versa. The design of Hetero Helix was motivated in particular by the requirements of robotics applications. Hetero Helix has been used successfully in an integration effort involving 27 CPUs in a heterogeneous network and a body of software totalling roughly 100,000 lines of code.

1. Introduction

Robot system architectures tend to be complex. They combine multiple sensor systems with effector control systems having many degrees of freedom, frequently using some intermediate form of higher-level "reasoning". They must integrate hard real-time systems with systems which cannot meet hard real-time deadlines. This complexity leads to an explosion in the amount of software necessary to perform what must seem to a casual observer to be trivial tasks.

Traditionally, work on robot system architectures has concentrated on principles for decomposing complex behavior into simpler behaviors, and has led to specific recommendations on the identity and functionality of the components in a finished system [1-4].

However, it is desirable to separate the issue of the precise identity of the components from the issue of how the components, regardless of their identities, are to communicate with one another. Previous work on communications systems for robotics [5,6] has generally used some concept of a centralized data structure for simplicity. In other cases [7,8], several different models of interprocess communication are supported, at the price of some complexity.

In this paper, we regard as arbitrary the identity and functionality of the components of a complex control system, and consider specifically interprocess communications mechanisms for integrating relatively large bodies of software into complete robotic systems. Our objectives are twofold: first, to provide an infrastructure suitable for interprocess and intersystem communications in a heterogeneous distributed-memory computing environment, and second, to make the properties of the communications system such that it is suitable for use in relatively large system integration efforts. To accomplish the second goal, we strive for simplicity.

Hetero Helix is a programming environment which supports interprocess communication in a heterogeneous network using a minimal suite of mechanisms. Specifically, Hetero Helix was developed to satisfy the software systems integration and interprocess/interprocessor

communications requirements of the U.S. Department of Energy's University Program in Robotics for Advanced Reactors [9].

This program has to date conducted three integrated technology demonstrations [10,11]. These demonstrations have required approximately 25,000, 50,000, and 100,000 lines of code respectively, and up to about 30 people (part-time) for development. Components include ultrasonic sensing and interpretation, computer vision [12], parallel processing, range image analysis [13], control of a 7-DOF redundant manipulator [14], control of an omnidirectional mobile platform [15], obstacle avoidance [16,17], path planning and path following, graphical user interfaces [18], and sensor-based force-reflecting teleoperation [19], among others. These demonstrations have made use of a relatively complex heterogeneous network of computers, including a Silicon Graphics IRIS 4D/70GT, a DEC Microvax, several VME-based common-bus multiprocessors, an Apple Macintosh, and an IBM-PC/AT based NCUBE hypercube multicomputer.

2. Motivation

The properties of Hetero Helix are motivated by three considerations. The first is the observation that complex control systems are of sufficient complexity to require a team for their development; they are too complicated to be developed by a single individual, or even a small team of closely interacting individuals. The second is the architectural requirements of various kinds of robotic control systems. The third is the communications technology required by heterogeneous distributed-memory systems.

Teamwork. Just as it is useful to adopt a simplified model of a complex computer system, it is useful to adopt a simplified model of the system development process [20]. A simple "project model", illustrated in Figure 1, provides a convenient mechanism for identifying the various activities in system development and the needs associated with each activity.

- *Design* details the method by which a desired complex behavior is to be generated by decomposing the whole behavior into components, each simple enough to be implemented by an individual or small team. Designers typically prefer to use abstract system models to avoid restricting developments to specific implementation idiosyncrasies.
- *Development* typically requires quite a bit of effort, and a correspondingly large staff. Information which must be shared by all parties in the development is therefore expensive. Furthermore, if the information is difficult to understand there is a high potential for error. Developers require a system which is conceptually straightforward and easy to use.
- *System integration* consists of testing each component independently and gradually assembling and testing the complete system. The output of each component must be accessible for inspection, it is necessary to be able to control each component individually, and it is necessary to be able to simulate a component's input in the absence of a completed input module.
- *Maintenance* includes not only finding and fixing bugs, but also extending and improving the system by adding capability to existing components and by adding new components.

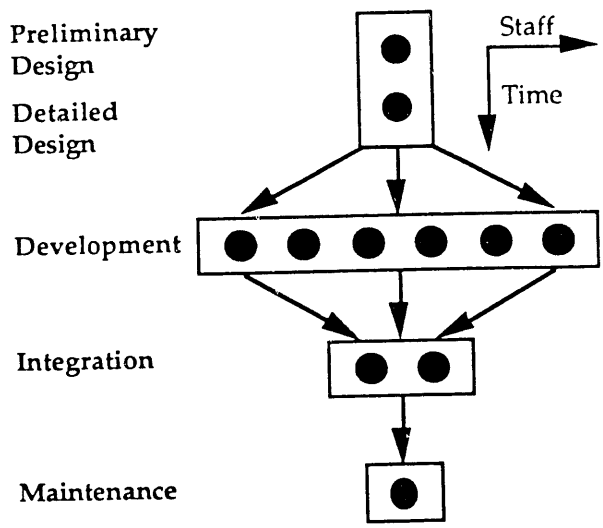

Figure 1. A simple model of the system development process. Staffing requirements are greatest during development, so the properties of the information which must be known to all parties in this stage is particularly crucial.

Architectural Requirements. A simplified requirements specification from a proof-of-concept demonstration for autonomous radiation surveillance [11] is illustrated in Figure 2. The scenario calls for a robot [21] initially at position (1) to navigate through a collection of a priori unknown obstacles until it reaches a position near (2). Once there, it uses a laser range camera to find 55-gallon drums. It selects a drum, moves to the

neighborhood of the drum, and re-acquires the drum using a CCD camera mounted on its manipulator arm. It then moves to a kinematically acceptable position (3) relative to the drum and uses a radiation sensor mounted on the end-effector to determine if there is radiation leaking from the drum.

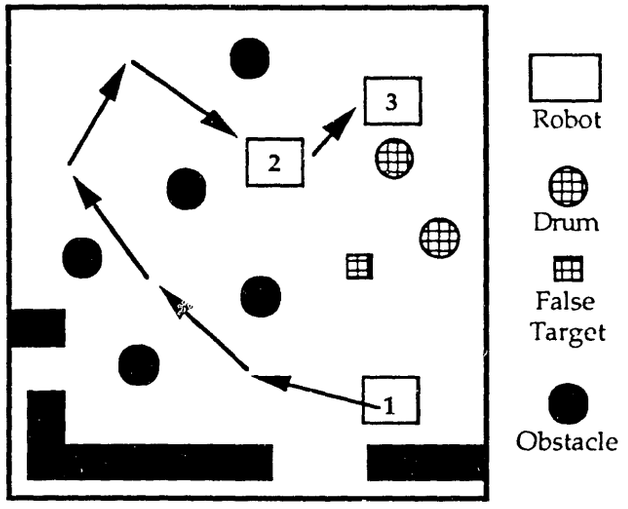

Figure 2. A simplified application scenario. A robot initially in position 1 is given a goal at position 2. Between positions 1 and 2 lie an unknown number of obstacles with unknown positions. At position 2, the robot uses a laser range camera to identify all of the 55-gallon drums in its field of view. It then chooses one of the drums for radiation surveillance using a detector mounted on its arm.

There are three distinct types of behavior which the robot must exhibit to accomplish this task.

The first type is exemplified by the sequence of events required to take the robot from position (2) to position (3). First it acquires a range image. Then it analyzes the image and chooses a drum. Then it travels to an appropriate position, and so forth. Each task must be accomplished in sequence, and consequently requires synchronization, but the exact time that it takes to execute each task is not critical.

The second type is exemplified by the process of travelling from one position to another. In this case, a collection of asynchronous processes consume and produce data essentially continuously. Figure 3 illustrates a diagram from the preliminary design of a navigation system which includes map-based obstacle avoidance based on ultrasonic data [16,17]. A "sonar_server" process continuously actuates sonars mounted on a ring encircling the robot chassis. A "sonar_mapper" process produces a bit-map of the robot's immediate neighborhood. An obstacle avoidance process examines this map, the robot's current position, and the goal position to produce a target for the low-level "wheel_controller". Each of these processes runs continuously and asynchronously. No interprocess synchronization is required.

The third type of behavior is exemplified by the process of scanning the drums for radioactivity. In this case we must coordinate the control of a relatively large number of joints and perform control calculations based on sampled data from optical encoders on the motors. All sensor input, servo controllers, and actuator drive outputs must be synchronized with a common clock [22].

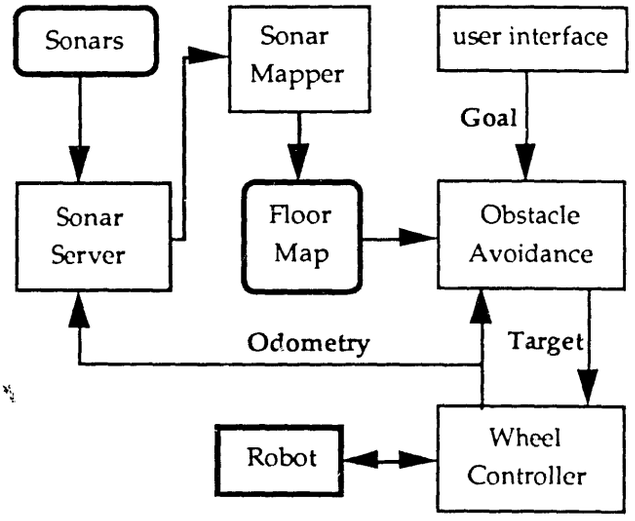

Figure 3. Simplified preliminary design of a navigation control system. Range data acquired by sonars are processed to create a map of the space surrounding the robot. The map is analyzed by an obstacle avoidance routine, which compares the current position of the robot to a user-specified goal and sends a target command to a wheel controller. The sonar server, sonar mapper, obstacle avoidance, wheel controller, and user interface in this design run concurrently, without interprocess synchronization.

These observations suggest that the programming environment must support interprocess and intersystem communication with or without synchronization and that mechanisms must be available to integrate hard real-time subsystems

with subsystems having less critical time constraints.

Heterogeneous systems. It is sometimes desirable to take advantage of the strengths of different computer systems to optimize performance of certain subsystems. For example, a graphical user interface usually requires special-purpose hardware for three-dimensional graphics but does not require a real-time operating system. In contrast, manipulator control requires real-time capabilities but has no particular need for graphics hardware.

The principal problems associated with communication in heterogeneous systems are two-fold: first, different computers run different operating systems which have different system calls, and second, different computers use different internal representations of data. It is desirable to insulate developer's code from the parochialism of particular operating systems and from the necessity to explicitly translate data from the internal format of one machine to another's.

3. Developer's Interface

Hetero Helix uses a modified blackboard metaphor for interprocess and intersystem communication. In this metaphor, we imagine that all of the data to be shared between processes, regardless of which processes produce or consume them, are posted on a single large blackboard covered with little yellow "Post-Its". Related data are collected together onto individual Post-Its. The Post-Its are concurrent read/exclusive write -- any process can read the data on any Post-It, but only one process can write to a Post-It.

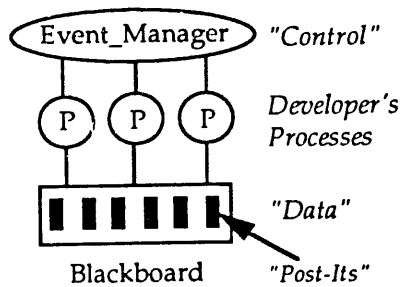

Figure 4. A simple abstraction of the system. Real-time and non-real-time subsystems (P) *communicate asynchronous data through a blackboard and synchronous control by sending events.*

Data on the blackboard are regarded as asynchronous -- any process may change the data on any of the Post-Its it owns without alerting any other process. Interprocess synchronization is accomplished by sending events. Events are small messages having a stereotyped format. Unlike data, which are written by one process but read by potentially many, events are process-to-process: any process sending an event specifies the identity of the process which is to receive it.

Figure 4 illustrates an abstraction of the system useful to designers and developers. Developer's processes, indicated with a "P", communicate data asynchronously by placing it into Post-It data structures on a blackboard. Control information, such as commands, are communicated by an event_manager. The event_manager has the responsibility of delivering the event to its proper destination.

The developer's interface to the communication system was designed to be as simple as possible. It consists of 10 functions, only two of which have more than one argument.

- *mem_attach()* returns a pointer to a list of Post-It addresses.
- *get_read_ok(Post-It)* increments a counting semaphore which records that the indicated Post-It is being read.
- *read_done(Post-It)* decrements the semaphore.
- *get_write_ok(Post-It)* waits until the read semaphore is clear, then sets a write semaphore blocking further reads.
- *write_done(Post-It)* clears the write semaphore. As a "side-effect" it sends an interrupt to a process which manages intersystem communications.
- *register_event(Process_Name)* identifies a process to the event manager.
- *resign_event()* notifies the event manager that a process is about to quit.
- *get_next_event(Event_Record*, Mode)* returns an event code, and optionally, its associated event record. The calling process may decide whether or not to suspend execution until an event is received by specifying blocking or non-blocking read in the Mode argument.
- *post_event(Code, Destination, Priority, Event_Record*)* sends an event to a process.
- *helix_broadcast(On/Off)* enables or disables intersystem communication by the calling process.

4. Systems View

Hetero Helix presents a simple blackboard-style interface to a heterogeneous distributed-memory network of workstations by simulating a shared memory [23]. In brief, this is accomplished by replicating the contents of the blackboard into each separate address space in the system and creating messages from individual Post-Its. The messages are broadcast to each address space in the network when the contents of a Post-It are updated; this is handled by interrupt-driven translation and message-passing routines hidden behind the call to "write_done()".

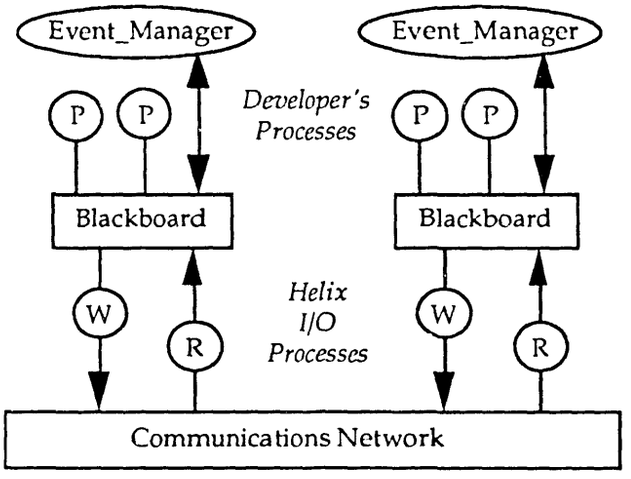

Figure 4. Mapping of the blackboard system onto a distributed heterogeneous system. The blackboard is replicated into each distinct address space. Communications processes (R and W) manage the translation of data into and out of the local representation employed on each machine. These processes are activated whenever a developer's process (P) updates the data in any of its Post-Its and calls "write_done()".

Figure 4 illustrates how the blackboard architecture is mapped onto a system consisting of two machines (additional machines replicate the structure). A copy of the entire blackboard is resident in the memory of each machine. Developers' processes communicate with each other through this blackboard. Whenever one of these processes executes a "write_done", an interrupt awakens a sleeping "write" process (W), which communicates with "read" processes (R) on the other machines in the network. These "read" processes recognize the incoming data and store it on the appropriate Post-It in the local copy of the blackboard.

There is an intimate relationship between events and the distributed blackboard system. If it is necessary for an event to travel from one system to another, it is placed in a Post-It reserved specifically for that purpose and broadcast to all the systems in the network. Upon receipt, each system may determine if the destination process is running on that system by referencing an event registry. If the destination process exists, the event is delivered normally, otherwise it is discarded. At the cost of some efficiency, this simple mechanism guarantees that all events are delivered to their proper destination.

The Hetero Helix "Compiler". The main problem with heterogeneous systems lies in the different internal representation of numbers which are used by different machines. The disastrous consequence of this inconsistency is that the actions associated with communicating a message are not independent of the contents of the message. For example, to communicate a floating-point variable it is not sufficient to know only the number of bytes it occupies -- one must translate the datum from the format recognized by the source machine into the format recognized by the destination. For complex data structures the creation and maintenance of the software necessary to translate each data structure can become a terrible burden.

Our approach to this problem is to use a "compiler" to automatically generate translation software given a description of the format of the data on the blackboard. That is, we imagine a machine which has an instruction set consisting of atomic instructions for translating variables from the internal representation of any particular machine into and out of a "lingua franca" common to all machines (we use IEEE 754 [24] for this purpose). The "compiler" inspects a file which describes the Post-Its on the blackboard and generates code in this instruction set which translates the data on the Post-Its into and out of this common representation.

Figure 5 illustrates the data structure Hetero Helix uses to represent the blackboard, a linked-list of linked-lists. The outermost list contains one link for each Post-It definition . Fields in each link record the name of the Post-It, its size, whether or not it is instantiated on the blackboard, and a pointer to another list. This innermost list contains a link for each variable in the Post-It definition. The variable name and it's storage class are recorded. In the event that the variable is an

array, its dimensionality and index limits are recorded as well. In the event that the variable is a complex data structure Hetero Helix stores a pointer to a list containing the details of its definition.

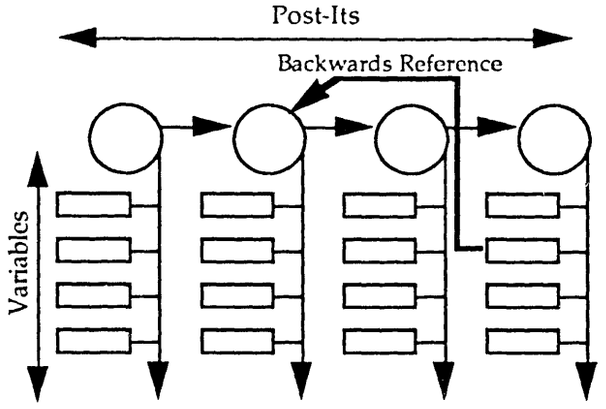

Figure 5. Illustration of the "linked-list of linked-lists" data structure employed by the Hetero Helix "compiler" for representing data on the blackboard. The outer list contains a link for each Post-It on the blackboard. The inner list contains a link for each variable on the Post-It. If one of these variables is a previously defined structure, a backwards reference is made to its definition.

The list-of-lists data structure is traversed several times to generate programs which (1) determine the appropriate placement for each Post-It on the blackboard and perform the appropriate initialization, (2) generate a data structure which provides to developers a mechanism for referencing data on the blackboard by name rather than by some difficult-to remember and easy-to-get-wrong index, and (3) translate Post-Its from the local data representation on each machine into a common representation and vice-versa.

Automatic code generation is useful in the development, integration, and maintenance phases of a project. For example, if the detailed design of the blackboard is in error for one reason or another (perhaps the designer forgot a variable), it is possible to fix all of the communications code in the system simply by running the code generator. The code which is produced is guaranteed to be correct the first time. This eliminates manual maintenance for a large body of what would otherwise be very expensive code. Other attractive benefits of automatic code generation include the automatic creation of a number of auxiliary routines for system diagnosis and maintenance.

5. Software Engineering Considerations

Hetero Helix addresses some (but certainly not all) of the differing needs of system designers, developers, integrators, and maintainers using either simple, but flexible, conceptual models or concrete delivered capability. In this section we make explicit the impact of various properties of Helix in each stage of the development process.

Design. **Hetero** Helix "flattens out" a heterogeneous, distributed memory environment by creating a fiction of a homogeneous system communicating via shared memory. Thus, to a first approximation, designer need not worry about the number of processors in the system, their relationship to one another, or the operating system software running on each. Hetero Helix makes no assumptions about master/slave relationships between hardware or software components. Hetero Helix assists in the detailed design by providing a simple, uniform protocol for the exchange of data. One must only specify the format of the data on each Post-It.

Development. Hetero Helix provides a small and simple function ensemble which executes all interprocess and intersystem communication services and a simple, uniform, easy-to-use syntax for referencing data. Consequently, error-prone communication between people is minimized. Developers are freed to concentrate on delivering capability in their component, rather than being distracted by complex interprocess communication protocols.

Integration. Hetero Helix provides a method by which system integrators can easily inspect the output of particular processes, simulate input to processes, and control individual processes for the purpose of assembling and testing subsystems. Data written to the blackboard stays there until it is overwritten. The data are available to any process, including in particular diagnostic scaffolding. A system integrator who understands what the correct results should be for a given situation can easily create diagnostic monitors which inform him of what the data actually are. Thus incorrect results are easy to detect. Furthermore, since all I/O in modules not tied directly to hardware is done through the blackboard, Hetero Helix makes it possible to move processes around in a

heterogeneous system without making changes to the source code.

Maintenance. Our policy of allowing only one process to write to any given location on the blackboard means that when incorrect data are detected, the offending process is identified immediately. Thus, errors can be isolated rapidly, and we can concentrate effort on a single process consisting of a few hundred to a few thousand lines of code, rather than having to search through the whole system for every bug.

Extending the system takes three forms: adding events to an existing process, extending the output of an existing process, and adding new processes. Adding events to a process involves no changes outside of that process except for a change to an event definitions file. Enhancing the output of a process or adding a new process involves changes to a blackboard definitions file. Helix imposes an implicit ordering of the Post-Its on the blackboard, and an implicit ordering of the data on each Post-It. Therefore, changes to the blackboard definitions file does not interfere with the operation of any existing program. Any process which wants to communicate using the new data elements has to be recompiled, but not other processes.

Additions to the blackboard definitions file requires that Hetero Helix generate a new communication system. Since Hetero Helix automatically generates all codes which depend on the definitions file, updating the communications system is quick, easy, and correct.

6. Specific Example

Figure 6 illustrates a specific hardware configuration to which Hetero Helix has been applied. The system consists of 27 processors in a heterogeneous network. The core of the system is a local area network based on ethernet. The blackboard is replicated in the address spaces of all of the machines directly connected to it. Events propogate to all the systems in the network, including those integrated using other communications technologies (e.g. bus adaptor, serial link). Mechanisms for distributing the blackboard using communications media other than ethernet are currently under study.

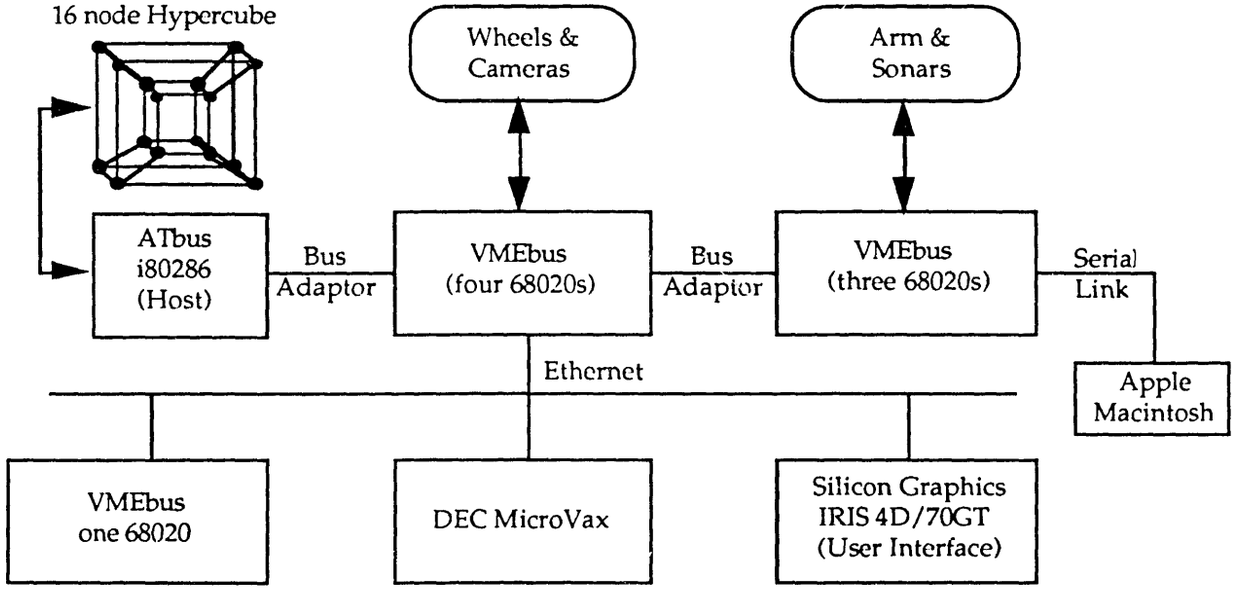

Figure 6. A system on which Hetero Helix has been implemented. Processes running on any CPU in this system (except the i80286) may communicate with one another using events. The blackboard is replicated in the address spaces of the Silicon Graphics, the VMEbus system with four 68020s, and either the DEC MicroVax or the VMEbus systems with one 68020.

7. Specializations for Robotics

Hetero Helix was designed for robotics applications, which are very often time sensitive. Perhaps the most objectionable property of the system from the perfomance perspective is the mandatory broadcast associated with each "write_done()", which may involve a substantial amount of overhead. There are three mechanisms by which this overhead may be avoided.

- The "helix_broadcast()" routine selectively turns broadcasting on and off on a process-by-process basis. Thus, a process may decide on its own how frequently to update local and global replicas of the blackboard.
- Alternatively, one may write an entirely separate process which periodically issues a "get_write_ok()/ write_done()" pair without changing the data on the Post-It. In this case, the process generating the data is never blocked.
- It is possible to ignore the communications protocol completely. Since Hetero Helix blackboard references use a direct addressing scheme (rather than going through intermediate software, as is done for example in Linda [25]), it is possible to "cheat" the system by referencing the blackboard without asking for read and write permission. This avoids even the relatively low overhead associated with the semaphore operations, and can be used to gain substantial time savings in instances where it is known that all consumers of certain data reside on the same system as their producer.

8. Discussion

Hetero Helix is a programming environment in which the identity and functionality of the various components in a complex control architecture are arbitrary. Hetero Helix supports synchronous and asynchronous distributed control systems by simulating shared memory on a heterogeneous network of computers. The key software technology in Helix is a "compiler" which analyzes the data structures in shared memory and automatically generates code which translates data representations from the internal format native to each machine into a format understood by all machines, and vice versa. Automatically generating the communications system for a heterogeneous network with the consequent increase in its reliability and maintainability is probably the system's most important property. Further developments will concentrate on providing support for a wider variety of communications hardware and for network reconfiguration.

The blackboard metaphor was chosen for this communications system not because the concept is new, but rather because it is not. Its familiarity makes it easy for designers to work with and for developers to understand. The simplicity of the Hetero Helix interface reduces the number of possible development-time errors.

One advantage of the Hetero Helix implementation of the blackboard is that it is not a physically centralized data structure. Replication makes it possible to develop subsystems which communicate through local address spaces only, in order to avoid saturating the intersystem communications hardware.

But one may legitimately wonder if the concept of a centralized data structure is a good long-term strategy. Modern scalable parallel computers almost always use some sort of message-passing hardware in order to avoid performance bottlenecks. Developing integration strategies which enable large teams to collaborate successfully in these kinds of computing environments promises to be challenging.

Acknowledgements

We would like to thank Wayne Manges for his constructive and insightful commentary at the outset of this effort, Reinhold Mann and Frank Sweeney for their support and encouragement, and Steve Johnston and Tom Heywood for their early work on the system. We would also like to thank the members of the DOE University Program in Robotics for Advanced Reactors for their patience and persistence. The submitted manuscript has been authored by a contractor of the U.S. Government under contract DE-AC05-84OR21400. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. The authors gratefully acknowledge the support of the U.S Department of Energy Office of Technology Support Programs and Office of Basic Energy Sciences.

References

1. R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," IEEE Journal of Robotics and Automation, (RA-2) 14-23, 1986.
2. J. S. Albus, H. G. McCain, R. Lumia, "NASA/NBS Standard Reference Model Telerobot Control System Architecture (NASREM)," NIST Technical Note 1235, NIST, Gaithersburg, MD, July, 1987.
3. T.L. Anderson, M. Donath, "Animal Behavior as a paradigm for developing robot autonomy." Robotics and Autonomous Systems (6) 145-168, 1990.
4. R.C. Arkin, "Motor schema based navigation for a mobile robot: an approach to programming by behavior." Proc. 1987 IEEE International Conference on Robotics and Automation, pp. 264-271.
5. S.Y. Harmon, "The ground surveillance robot (GSR): An autonomous vehicle designed to transit unknown terrain," IEEE Journal of Robotics and Automation (RA-3) 266-279, 1987.
6. C. Thorpe, M. H. Herbert, T. Kanade, S. A. Shafer, "Vision and navigation for the Carnegie-Mellon Navlab," IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI-10) 362-373, 1988.
7. D.B. Stewart, D.E. Schmitz, P.K. Koshla, "Implementing real-time robotics systems using CHIMERA II", Proc. 1990 IEEE International Conference on Robotics and Automation, pp. 598-603.
8. R. Chatila, R.F. Camargo, "Open architecture design and inter-task/inter-module communications for an autonomous mobile robot," IEEE International Workshop on Intelligent Robots and Systems, pp. 717-721, 1990.
9. F. J. Sweeney, "ORNL Research in the DOE University Program in Robotics for Advanced Reactors," American Nuclear Society Annual Meeting, Nashville, Tenn., June 10–14, 1990.
10. D.B. Reister, J.P. Jones, P.L. Butler, M. Beckerman, F.J. Sweeney, "Demo 89 - The initial experiment with the Hermies-III robot." Proc. 1991 IEEE International Conference on Robotics and Automation, pp. 2562-2567.
11. F.J. Sweeney, M. Beckerman, P.L. Butler, J.P. Jones, D.B. Reister (1991) "Application of autonomous robotics to surveillance of waste storage containers for radioactive surface contamination." Proc. AI '91: Frontiers in Innovative Computing for the Nuclear Industry, Jackson, Wyoming, September 15-18, 1991.
12. C. Chen, M. M. Trivedi, C. R. Bidlack, "Design and implementation of an autonomous spill-cleaning robot," *Proc.* Applications of Artificial Intelligence XIII, 1990.
13. J. C. Sluder, C. R. Bidlack, M. A. Abidi, M. M. Trivedi, J. P. Jones, F. J. Sweeney, "Range image-based object detection and localization for the HERMIES-III mobile robot," Proc. Applications of Artificial Intelligence IX, pp. 642-652, 1991.
14. R. V. Dubey, J. A. Euler, S. M. Babcock, and R. L. Glassell, "Real Time Implementation of a Kinematic Optimization Scheme for Seven-Degree-of-Freedom Redundant Robots with Spherical Wrists," The American Control Conference, Atlanta, Ga., June 15–17, 1988.
15. D.B. Reister, "A new wheel control system for the omnidirectional Hermies-III robot." Proc. 1991 IEEE International Conference on Robotics and Automation, pp. 2322-2327.
16. J. Borenstein, Y. Koren "The Vector-Field Histogram - Fast Obstacle Avoidance for Mobile Robots," *IEEE Transactions on Robotics and Automation* (RA-7), 278-288, 1991.
17. J. Borenstein, Y. Koren "Histogrammic in-motion mapping for mobile robot obstacle avoidance." *IEEE Transactions on Robotics and Automation* (RA-7), 535-539, 1991.
18. C. Crane, R. Vora, J. Tulenko, G. Dalton, "Model Simulation for robotic control and intelligence." American Nuclear Society Third Topical Meeting on Robotics and Remote Systems, Charleston, SC, 1989.
19. J.T. Lovett, P. Bevill, "A universal bilateral manual controller utilizing a unique parallel architecture," *Transactions of the American Nuclear Society* (61) 409, 1990.
20. F. McGarry, J. Page, S. Eslinger, V. Church, P. Merwarth, "Recommended approach to software development." NASA Software Engineering Laboratory Technical Memorandum SEL-81-205, 1983.
21. C. R. Weisbin, B. L. Burks, J. R. Einstein, R. R. Feezell, W. W. Manges, D. H. Thompson, "HERMIES-III: A step toward autonomous mobility, manipulation and perception," Robotica (8) 7-12, 1990.
22. P.L. Butler, "An integrated architecture for modular control systems," Robotics and Autonomous Systems, 1991, in press.
23. B. Nitzberg, V. Lo, "Distributed shared memory: A survey of issues and algorithms." IEEE Computer (24) 8, 52-60, 1991.
24. IEEE. "IEEE standard 754-1985 for binary floating-point arithmetic." Reprinted in SIGPLAN 22, 2, 9-25, 1987.
25. N. Carriero, D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *A C M Computing Surveys* (21) 323-357, 1989.

DATE
FILMED
4/6/92