Building Distributed Rule-Based Systems Using the AI Bus

Dr. Roger D. Schultz and Iain C. Stobie
Abacus Programming Corporation

Abstract

The AI Bus software architecture was designed to support the construction of large-scale, production-quality applications in areas of high technology flux, running on heterogeneous distributed environments, utilizing a mix of knowledge-based and conventional components. These goals led to its current development as a layered, object-oriented library for cooperative systems.

This paper describes the concepts and design of the AI Bus and its implementation status as a library of reusable and customizable objects, structured by layers from operating system interfaces up to high-level knowledge-based agents. Each agent is a semi-autonomous process with specialized expertise, and consists of a number of knowledge sources (a knowledge base and inference engine). Inter-agent communication mechanisms are based on blackboards and Actors-style acquaintances. As a conservative first implementation, we used C++ on top of Unix, and wrapped an embedded Clips with methods for the knowledge source class. This involved designing standard protocols for communication and functions which use these protocols in rules. Embedding several Clips objects within a single process was an unexpected problem because of global variables, whose solution involved constructing and recompiling a C++ version of Clips. We are currently working on a more radical approach to incorporating Clips, by separating out its pattern matcher, rule and fact representations and other components as true object oriented modules.

1. Introduction

The AI Bus is a software architecture and toolkit which supports the construction of large-scale, production-quality cooperating systems in areas of high technology flux. It was first developed as an approach to integrating the Space Station software, and more recently has been applied to the Advanced Launch Systems project (ALS). Both applications share requirements of a long life-time, during which new technological advances should be seamlessly incorporated, and high degrees of autonomy. These two classes of requirements - the software engineering need for flexible methods for combining heterogeneous components, and the functional need to coordinate a mix of knowledge-based and conventional systems - led to the development of the AI Bus as a layered, object-oriented, distributed architecture.

This paper describes the concepts and design of the AI Bus and its current implementation as a Unix C++ library of reusable objects. After an introduction to distributed processing and a discussion of the facilities needed to build cooperating systems, we present the mechanisms provided by the AI Bus for these facilities. Particular emphasis is placed on supporting high-level models of cooperation and problem-solving, implemented via semi-autonomous agent processes with knowledge-based communication and control. Finally we describe our approach to using Clips as a common knowledge representation language for the prototype.

2. Overview of Distributed Cooperative Systems

A distributed system may be characterized as a collection of separate processes together with an interaction medium. This separation and the interaction medium may be physical, as in processors connected by a network, or logical, as in modules with semantically disparate representations. Although developments in the last fifteen years have taken advantage of hardware advances by distributing data and processing, the control has remained centralized in master-slave relationships. Machines are now "talking" to one another, but the question for cooperative systems is deciding what to say, when, and by

whose authority. Just as humans form organizations in order to function more effectively - the whole is greater than the sum of the parts - the promise of cooperative systems is that they can tackle problems beyond the capabilities of current architectures.

Cooperative systems use advances in distributed processing - algorithms for load balancing, efficient network routing, error recovery procedures, synchronization mechanisms, etc. - but build on them by treating the distribution as part of the problem solving which needs to be represented and reasoned about. For example, a distributed database should appear coherent to its users, but maintaining its global consistency is impossible without synchronizing transactions, and this may be prohibitively slow. The promise of cooperative systems is that such problems are amenable to techniques of modelling the users' goals and plans, handling uncertainty and inconsistency gracefully, and adaptively allocating tasks and resources (Ref. [1,2]).

If an agent is to help another it must have a way to represent that agent's goals and plans, if it is to receive help it must know which agents are able to provide assistance and hence must model their abilities and resources, and if it is simply interested in avoiding conflict it must be aware of their planned use of shared resources. Thus facilities are needed for modelling capabilities and interests, above simple interface specifications, and knowledge-based protocols for negotiation. Some approaches to realizing these goals are (Ref [3]):

- Distributed Object-Oriented Systems (DOOS): A natural way to model cooperative systems uses the object-oriented paradigm of autonomous modules communicating via messages. Extending this paradigm to distributed environments involves difficult problems of several threads of control and no single shared space of objects. (Ref. [4,5,6)
- Blackboards: In contrast to the message-passing model of DOOS, blackboards are an organizational mechanism whereby agents share their current problem solving state. (Ref. [7])
- Integrative Frameworks: Systems which combine a number of different mechanisms to support various paradigms for developing and experimenting with large scale applications. (Ref. [8,9,10]).

3. Facilities Provided by the AI Bus for Building Cooperative Systems

3.1 Overview of Goals and Features

The AI Bus is an integrative framework for building cooperating systems with the following requirements:

- Technology Transparency: the architecture is open to allow integration of future advances and is portable across disparate platforms.
- High Performance: the emphasis is on production quality, rather than experimentation.
- Support multiple coexistent problem solving paradigms: DOOS, blackboards, expert systems.
- Standard interfaces for combination of components and communication between subsystems.
- Mixed conventional and AI Approaches: through standard interfaces; included is the ability to incorporate off the shelf commercial tools.
- Support for verification and validation: integrated tools include dynamic audit probes (which can feed diagnosis and repair modules) and static compile-time checking of interfaces.

3.2 Software Engineering Principles

The components are divided into layers based on their abstraction level: at the bottom are the physical entities, then the operating system components, then conventional tools such as databases and user interfaces, followed by knowledge-based tools such as inference engines, and at the top are generic applications such as diagnosis shells which simply need to be customized for a specific application. Each layer provides services to higher layers; since the internal details of a layer are hidden from others, software changes are localized and modules are easily replaceable; for performance reasons a layer is permitted to call a lower non-adjacent layer rather than strictly stepping through the intermediaries. The AI Bus is defined as a set of object classes, and implemented as a class library, which again enables the

internal implementation of objects to be hidden from other objects. Off the shelf components can be integrated by wrapping them in a suitable interface, but clearly the degree of support they receive from other AI Bus services is proportional to their "white-box" nature. The layers and representative object classes are illustrated in Figure 1, while the inheritance between a few classes is shown in Figure 2.

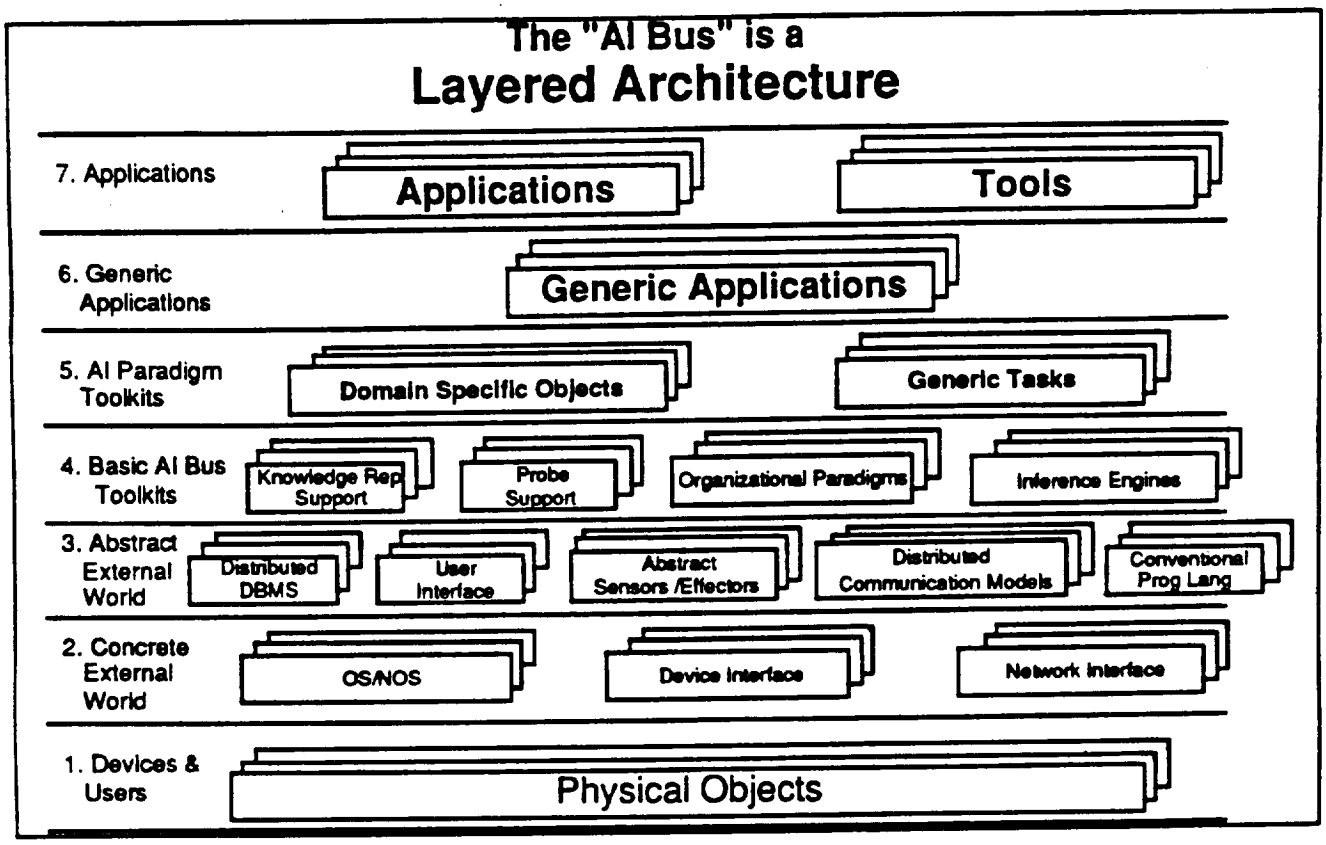

Figure 1. The layers of the AI Bus

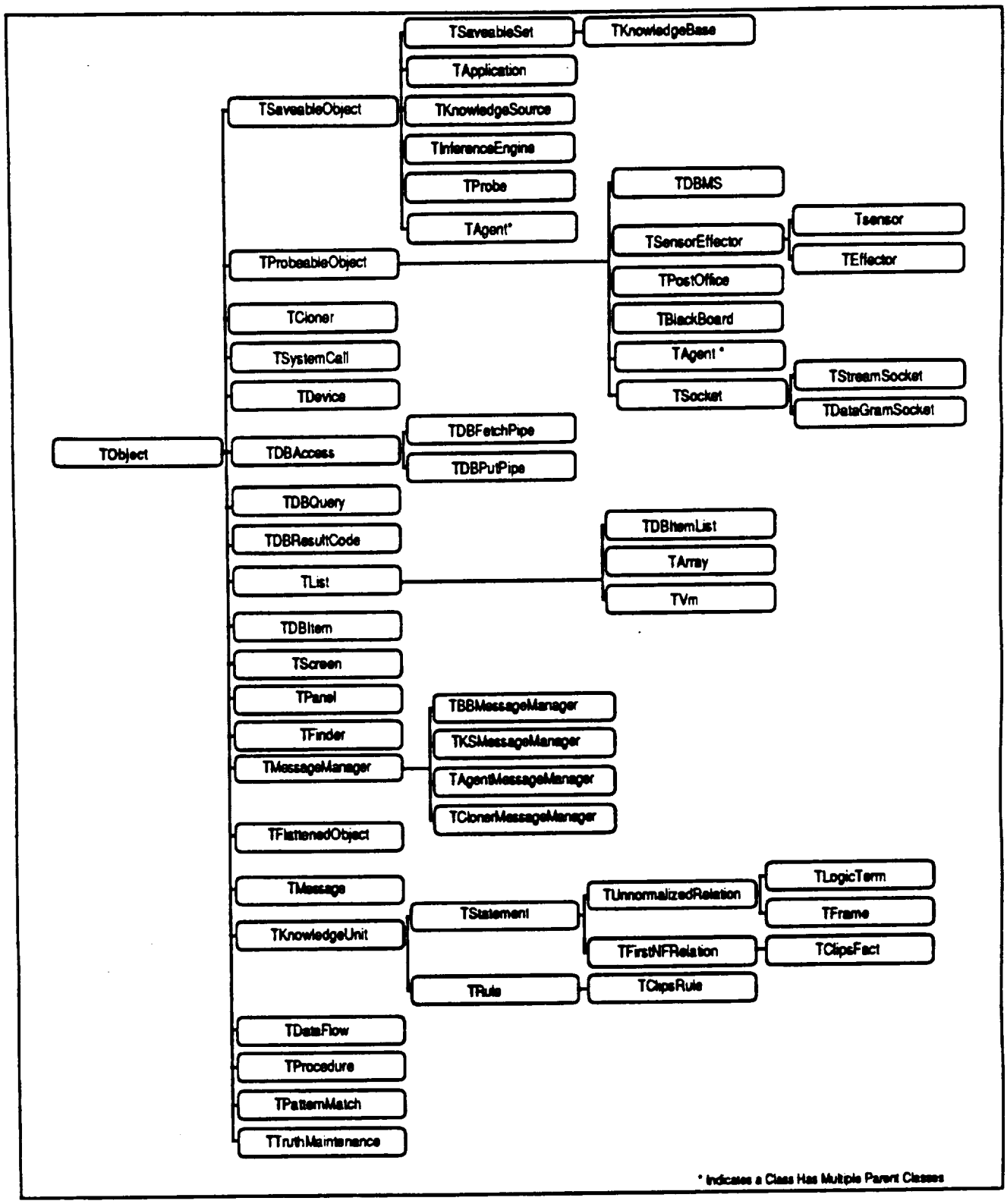

Figure 2. A Subset of the AI Bus Class Hierarchy

3.3 Probes, Messages, Agents and Organizations

3.3.1 Probes and Event-Driven Programming

The AI Bus follows the distributed object oriented model of interaction between software modules, here considered to be loosely-coupled agents. This not only supports the above software engineering principles, but also the open, continuous processing that is a characteristic of cooperative systems. Whereas event handlers in conventional systems, such as X Windows or database transactions, are invoked from a dispatch table using simple masks or triggers, the AI Bus extends this paradigm in its Probe object (Ref. [11,12]). A probe is activated based on matching patterns of events and conditions and routes information about subsystem activity to interested parties which can install and modify them dynamically. A probe's history can be used to maintain partial matches for efficiency (e.g. in the blackboard), its priority can be used to order the actions of several probes. A standard event, condition and action language allows the evaluation and interpretation of probes to be implemented by the probed object - a class of probeable objects is specified, and includes databases, network communication, blackboards and agents; there are corresponding subclasses of probes.

Probes can be used to support validation in a testbed environment and to monitor resource usage and each other. Since they are implemented by the probed object and installed by request, this access does not violate the secure boundaries of active objects. A subclass of probes called abstract sensor/effectors can be used in hierarchical process control applications - like probes they provide data, retain state and do some filtering, but in addition they recognize alarm situations and provide direct pathways between each other for fast response.

3.3.2 Communication Substrate

A layer of services exists between the operating system and the programming tools which allows the developers to concentrate on problem-solving rather than worrying about actual physical locations. Of course, for some applications, physical parameters are part of the problem definition (e.g. communication delays, noise and failures) and so are available for querying. Each agent has a Post Office object, which queues incoming messages and permits addressing by name, rather than location. The Post Office uses a distributed Finder object, which keeps track of the addresses of active objects and maps them to their globally unique names. Furthermore, agents can advertise certain attributes (see later section) which are also registered with the Finder and permit communication by knowledge rather than just syntactic names.

The interaction medium is the message, the glue which enables the transfer of data and control between the agents. A message contains fields which identify the sender and receiver, an object (such as a question or answer) an optional time tag and list of attributes, which may include its expiration date or other application-specific information. Control is passed via messages which represent remote procedure calls - they are intercepted by an agent's Message Manager, which is responsible for converting messages to procedures, and keeps a queue of questions received together with their askers (for subsequent direction of replies). Remote procedure calls by default are asynchronous - the caller doesn't block and wait for its completion - but may be synchronous if required. The question of whether the receiving agent blocks until it processes the request depends on the organization used: if the agent does, it is under the control of the sender (a client-server relationship), if not it is autonomous. Of course, requests to lower-level services (such as a database manager) are processed synchronously - only high-level agents can own a thread of control.

3.3.3 Agent

The agent is the fundamental active entity in the AI Bus, encapsulated as an object which communicates by messages. Currently an agent and its message manager occupy a Unix process, so its boundary exists not only as a software object but is also enforced at the operating system level. An agent is defined as a collection of knowledge sources and an organization; these knowledge sources may be implemented as expert systems (an inference engine and a knowledge base) or a conventional system - just so long as the specified interface is

followed. Each knowledge source has a list of capabilities and interests - which match questions it can answer and information it would like to be told - the agent advertises these attributes with the Finder and keeps a cache of other agents' capabilities and interests for subsequent communication.

An agent's specification thus permits implementation along several sizes of granularity. Internally, it can be a whole organization of problem solvers, or just a simple procedural program. It has a scheduler component for control of its knowledge sources and is not necessarily serial (it may be realized as one or several processes or threads). Its state may be dormant or active, but currently most agents are eternally vigilant or waiting for a reply. For efficiency reasons in Unix-like environments a large grain may be preferred, and this can be used at the next layer up as a generic task - an agent which is a specialist in one area of problem solving (Ref. [13]).

An agent's capabilities and interests represent a model of its goals, plans, abilities and needs that other agents can use for cooperation. An agent can choose not to cooperate by not advertising this model, but in general they can build up more extensive models of each other by starting with the originally advertised capabilities and interests and then learning from experience by caching results: for example, two agents may have a capability to do arithmetic, but by trying each the faster one is identified and will be preferred in future requests. An agent can have a reflective ability by installing probes in itself (for example, to measure the number of rules fired by a knowledge source's inference engine); this allows it to monitor its progress and interrupt if necessary. The combination of agents into a cohesive problem-solving team is achieved by creating an organization. One example of the internal organization of a complex agent is illustrated in Figure 3.

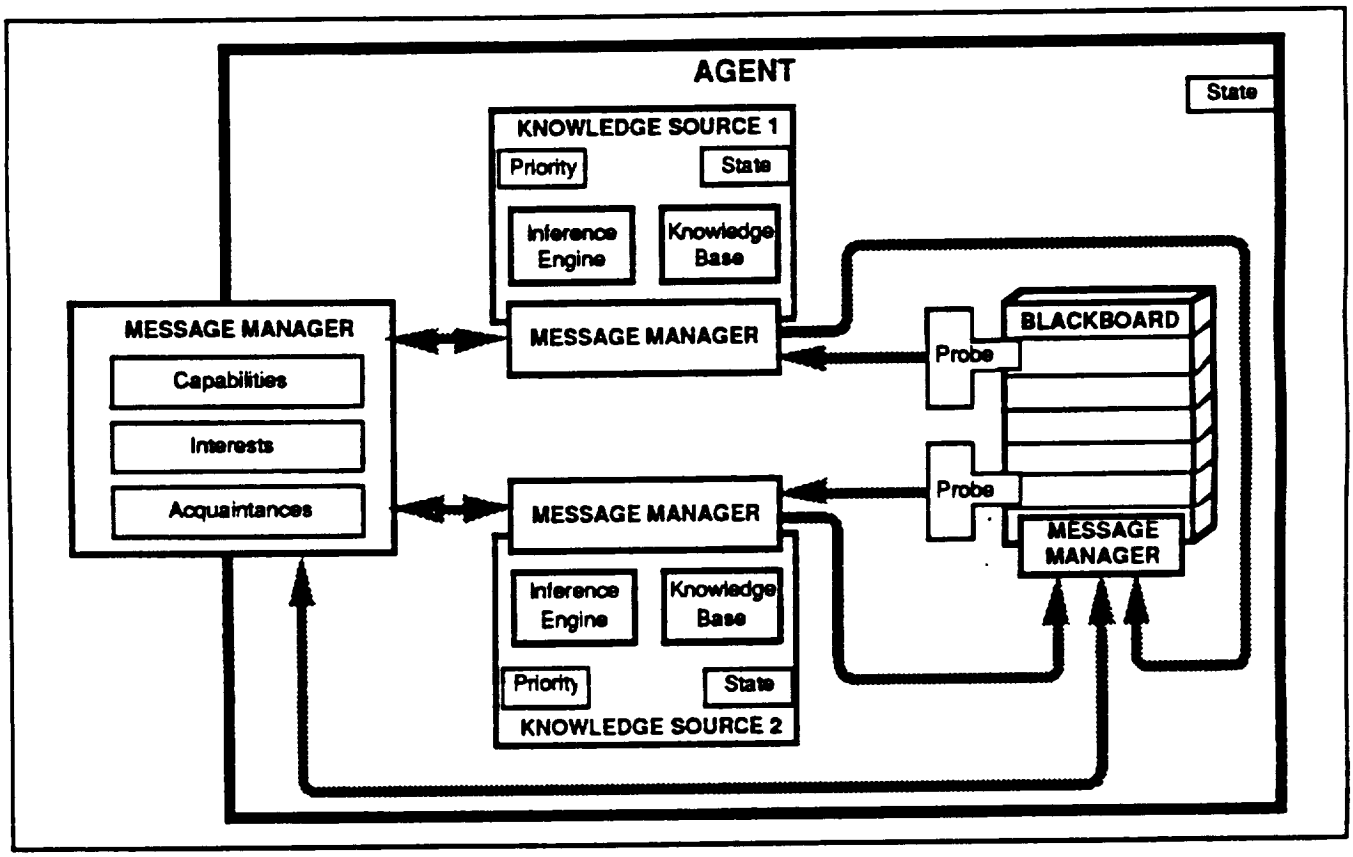

Figure 3. An Example of an Agent Composed of Several Layer 4 Objects

3.3.4 Organization

An organization is simply a collection of agents who know each others' capabilities and interests - this is an implicit specification by knowledge existing in each agent. In contrast to structural definitions of organizations, this model is adaptive, since agents can compute who knows how to answer a question it cannot itself process, and thereby new relationships form within the organization. One agent can be programmed to act as a manager, who delegates work to other agents according to their advertised capabilities, monitors their progress using probes and adjusts their position in the organization .

A final method to combine agents is more indirect, by sharing access to a blackboard. A blackboard is realized in the AI Bus as a restricted subclass of agent - it is a passive server which is interested in everything (or at least whatever it is programmed for). Agents post information on the blackboard by sending it messages, they install probes on it to gather information resulting from matching events plus several current and historical conditions. A blackboard is thus a semi-permanent communication space, but also acts as a mechanism for loosely-coupled organization whereby several agents can combine partial results without repeated inter-agent communication. It is more than a global database, in that the probes' histories provide a short-term memory and record of partial matches, so that new additions and requests can be processed quickly (in the style of the Rete algorithm for rule-based systems); in contrast, database queries are processed one at a time. This is an object-oriented version of the blackboard concept, and it is important to contrast it with blackboard systems which contain a centralized scheduler in control of the serial execution of agents: in the AI Bus the agents are autonomous. Although logically centralized, a blackboard may be physically distributed for performance reasons: in this case, consistency must be maintained using techniques (e.g. multiple copies, deadlock avoidance) borrowed from distributed databases. An illustration of the different methods of communication and cooperation is shown in Figure 4.

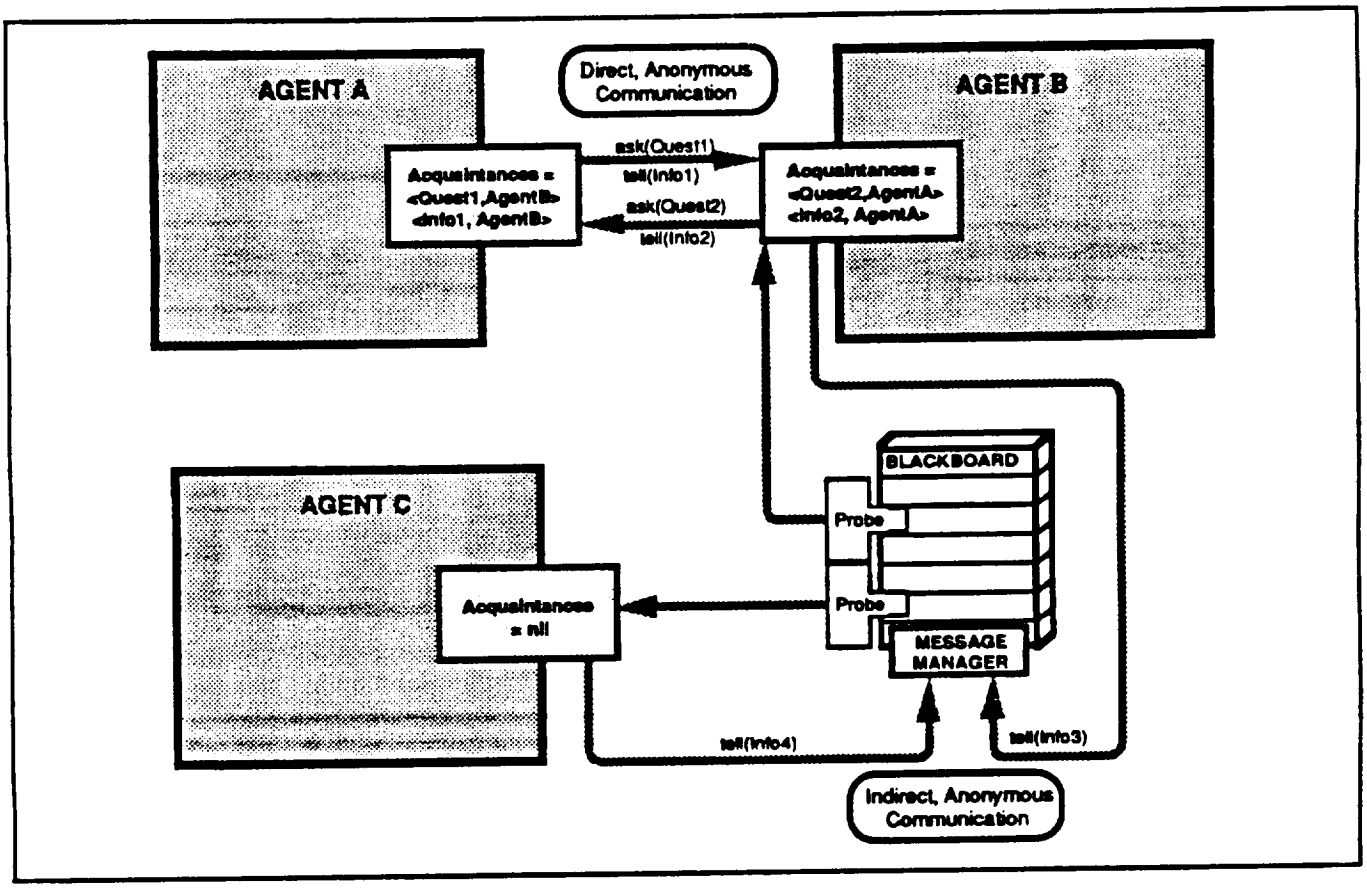

Figure 4. Agents Communicate directly with their Acquaintances and indirectly via Blackboards

4. Development of the AI Bus

The design of the AI Bus was first summarized in a set of abstract-data-type class specifications, intentionally kept language-independent in order to avoid restricting the design. See Figure 5.

Class
TAgent

Description
A heterogeneous single-task semi-autonomous expert

Superclass
TSaveableObject, TProbeableObject

Components
Set of KnowledgeSources
Tables of capabilities and interests
Organizational Paradigm Objects
State (Active/Inactive)

Functions
Install(List of Knowledge Sources)
Action : initializes data structures and constructs capabilities and interests
Remove: nil -> nil
Action: removes itself from memory, killing its processes
Ask (Message)
Action : passes message to knowledge source with the capability. Caller may or may not block
Answer (Message)
Action : passes message to knowledge source which asked the question
Tell (Message)
Action : passes message to the knowledge source with the interest
Run(Amount)
Action : runs each knowledge source for an amount of time, cycles, or until condition
Send<Question|Answer|Tell> (Message))
Action : sends message to the remote agent, using Finder (if Question or Tell) or the sender's address (if Answer)

Policy
1) For remote access, these methods will be called from an attached message manager
2) The message manager is responsible for routing messages according to capabilities, interests and acquaintances
3) Inter-agent communication can be goal-driven (Ask) or event-driven (Tell)

Figure 5. Class Description for an Agent

4.1 Initial Implementation Approach

For the implementation, we chose C++ and Unix because of the performance benefits of a relatively low-level language and its wide availability: a fundamental goal was to build a production quality system, not an experimental testbed. For the common knowledge representation language (a KnowledgeUnit class) we chose Clips because it is distributed with source code and hence is amenable to customization. Message passing between Clips agents was easily accomplished by writing three user-defined C++ functions (aibus_ask, aibus_tell, aibus_answer) that are called from the right hand side of a Clips rule and in turn invoke the encapsulating agent's methods (Figure 5) to interface with remote objects. The communication services were built on top of the RPC protocol.

This choice of implementation tools resulted in the compromise that an agent could only contain one knowledge source: a C++ object resides in one process, but having several Clips instantiations in one process is impossible because of its global variables. Furthermore, we had hoped to isolate out the inference engine components (pattern match, agenda scheduling, etc.) from the Clips source code for reuse in other Layer 4 objects such as probes and blackboards; however C++'s strongish typing caused problems in handling free-style C, even with the help of Abacus' automatic translation system, MetaPack. As a result we had to write our own procedures for these purposes and just treat Clips as a black-box KnowledgeSource object rather than a composite object (see Ref. [14] for the approach used in the Joshua system).

4.2 Current Implementation Direction

We are currently pursuing the direction outlined above, of decomposing the functionality of a Clips based inference engine into object-oriented modules. Agents could then have several Clips components, rules could inherit conditions and actions from other rules, rule-bases could inherit rules from other rule bases and the distinction between rule-based and frame-based languages would disappear. For hard real-time situations, the Rete net's good average-time but unpredictable worst-time performance is unsuitable and alternative implementations are necessary. For example, linear searching with compiled-out pattern matching (e.g. L*Star, Ref. [15]), or search algorithms like iterative deepening which always maintain the best-solution-so-far.

We are also working on incorporating non-linear fact and pattern representations (e.g. Prolog's recursive structures), and providing more support for probe access to AI Bus objects, especially for dynamic validation. At the cooperative systems level, we are experimenting with negotiation protocols, and providing agents with learning capabilities.

References

[1] Lesser,V. Corkill,D. *The Distributed Vehicle Monitoring Testbed.* AI Magazine, Fall 1983

[2] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. *Coherent Cooperation Among Communicating Problem Solvers.* IEEE Transactions on Computers, C-36:1275-1291, 1987

[3] Alan H. Bond and Les Gasser. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[4] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986

[5] Yutaka Ishikawa and Mario Tokoro. *Orient84/K: An Object Oriented Concurrent Programming Language for Knowledge Represetnation* in Object Oriented Concurrent Programming, Yonezawa & Tokoro, eds, MIT Press, 1987

[6] A. Yonezawa, J-P. Briot, E. Shibayama. *Object-Oriented Concurrent Programming in ABCL/1.* in [3]

[7] Penny Nii. *Blackboard Systems.* AI Magazine Volume 7, nos. 3 and 4

[8] R. Bisiani, F. Alleva, A. Forin, R. Lerner, M. Bauer. *The Architecture of the Agora Environment* in Distributed Artificial Intelligence, Michael N. Huhns, Ed., Morgan Kaufman, 1987

[9] Lee D. Erman, Jay S. Lark, and Frederick Hayes-Roth. *ABE: An Environment for Engineering Intelligent Systems.* IEEE Transactions on Software Engineering 14(12), December 1988

[10] Les Gasser, Carl Braganza, Nava Herman. *Implementing Distributed AI Systems Using MACE.* in [3]

[11] Roger D. Schultz and A. Cardenas. *An Approach and Mechanism for Auidtable and Testable Advanced Transaction Processing Systems.* IEEE Transactions on Software Engineering, SE-13 (6), June 1987

[12] Roger D. Schultz and A. Cardenas. *An Expert System Shell for Dynamic Auditing in a Distributed Environment.* ACM SIGSAC '87 Conference Proceedings

[13] B. Chandrasekaran. *Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design.* IEEE Expert, Fall 1986

[14] S. Rowley, H. Shrobe, R. Cassels, W. Hamscher. *Joshua: Uniform Access to Heterogeneous Knowledge Structures*. AAAI-87

[15] T. Laffey, P. Cox, J. Schmidt, S. Kao. J. Read. *Real-Time Knowledge-Based Systems*. AI Magazine, Spring 1988