A Hierarchical Distributed Control Model for Coordinating Intelligent Systems

Richard M. Adler
Symbiotics, Inc.
875 Main Street
Cambridge, MA 02139

ABSTRACT

This paper describes a hierarchical distributed control (HDC) model for coordinating cooperative problem-solving among intelligent systems. The model was implemented using SOCIAL, an innovative object-oriented tool for integrating heterogeneous, distributed software systems. SOCIAL embeds applications in "wrapper" objects called Agents, which supply predefined capabilities for distributed communication, control, data specification and translation. The HDC model is realized in SOCIAL as a "Manager" Agent that coordinates interactions among application Agents. The HDC-Manager: indexes the capabilities of application Agents; routes request messages to suitable server Agents; and stores results in a commonly accessible "Bulletin-Board". This centralized control model is illustrated in a fault diagnosis application for launch operations support of the Space Shuttle fleet at NASA, Kennedy Space Center.

Keywords: distributed artificial intelligence, systems integration, hierarchical distributed control, intelligent control, cooperative problem-solving

INTRODUCTION

Knowledge-based systems are helping to automate important functions in complex problem domains such as operations and decision support. Successful deployment of intelligent systems requires: (a) integration with existing, conventional software programs and data stores; and (b) coordinating with one another to share complementary knowledge and skills, much as people work together cooperatively on related tasks. These requirements are difficult to satisfy given existing AI technologies. Current knowledge-based systems are generally single-user, standalone systems based on heterogeneous data and knowledge models, development languages and tool shells, and processing platforms. Interfaces to users, databases, and other conventional software systems are typically custom-built and difficult to adapt or interconnect. Moreover, intelligent systems developed independently of one another tend to be ignorant of information resources, problem-solving capabilities, and access protocols for peer systems.

SOCIAL is an innovative collection of object-oriented tools designed to alleviate these pervasive integration problems [Ad90b]. SOCIAL provides a family of "wrapper" objects, called *Agents*, that supply predefined capabilities for distributed communication, control, data specification and translation. Developers embed programs within Agents, using high-level, message-based interfaces to specify interactions between programs, their embedding Agents, and other application Agents. The relevant Agents transparently manage the transport and mapping of specified data across networks of disparate processing platforms, languages, development tools, and applications. SOCIAL's partitioning of generic and application-specific behaviors shields developers from network protocols and other low-level complexities of distributed computing. More important, the interfaces between applications and SOCIAL Agents are modular and non-intrusive, minimizing the number, extent, and cost of modifications necessary to re-engineer existing

systems for integration. Non-intrusiveness is particularly important in mission-critical space and military applications, where alterations for integration entail stringent validation and verification testing.

This paper focuses on a specialized "Manager" Agent that realizes a hierarchical distributed control (HDC) model on top of SOCIAL's basic integration services. The SOCIAL HDC-Manager Agent coordinates the activities of Agents that embed independent knowledge-based and conventional applications relating to a common domain such as decision or operations support. Such centralized control models are important for managing distributed systems that evolve over time through the addition of new applications and functions. Centralized control is also important for organizing complex distributed systems that display not only small-scale, one-to-one relationships, but also large-scale structure, such as clustering of closely related subsets of application elements.

The HDC-Manager Agent's coordination functionality derives from a set of centralized control services including: maintaining an index knowledge base of the capabilities, addresses, and access message formats for application Agents; formatting and routing requests for data or problem-solving processing to suitable server Agents; and posting request responses and other globally useful data to a commonly accessible "Bulletin-Board".

The next section of the paper reviews the overall architecture and functionality of SOCIAL. Subsequent sections describe the structure and behavior of the HDC-Manager Agent and illustrate its application in the domain of launch operations support for the Space Shuttle fleet at NASA, Kennedy Space Center. Specifically, a HDC-Manager Agent coordinates the activities of standalone expert systems that monitor and isolate faults in Shuttle vehicle and Ground Support systems. The cooperative problem-solving enabled by the HDC-Manager produces diagnostic conclusions that the applications are incapable of reaching individually.

OVERVIEW OF SOCIAL

The central problems of integrating heterogeneous distributed systems include:

- communicating across a distributed network of heterogeneous computers and operating systems in the absence of uniform interprocess communication services;
- specifying and translating information (i.e., data, knowledge, commands), across applications, programming languages and development shells with incompatible native data representations;
- coordinating problems-solving across applications and development tools that rely on different internal models for communication and control.

SOCIAL addresses these issues through a unified collection of object-oriented tools for distributed communication, control, data (and data type) specification and management. Developers access the services provided by each tool through high-level Application Programming Interfaces (APIs). The APIs conceal the low-level complexities of implementing distributed computing systems. This means that distributed systems can be developed by programmers who lack expertise in areas such as interprocess and network communication (e.g., Remote Procedure Calls, TCP/IP, ports and sockets), variations in data architectures across vendor computer platforms, and differences among data and control interfaces for standard development tools such as AI shells. Moreover, SOCIAL's high-level development interfaces to distributed services promote modularity, maintainability, extensibility, and portability.

The overall SOCIAL architecture is summarized in Figure .1. SOCIAL's predefined distributed processing functions are bundled together in objects called *Agents*: Agents represent the active computational processes within a distributed system. Developers assemble distributed systems by:

(a) selecting and instantiating Agents from SOCIAL's library of predefined Agent classes; and (b) embedding individual application elements such as programs and databases within Agents. Embedding consists of using the APIs for accessing SOCIAL's distributed processing capabilities to establish the desired interactions between applications, their associated wrapper Agents, and other application Agents. New Agent subclasses can be created through a separate development interface by customized or combining services in novel ways to satisfy unique application requirements. These new Agent types can be incorporated into SOCIAL's Agent library for subsequent reuse or adaptation. The following subsections review the component distributed computing technologies used to construct SOCIAL Agents.

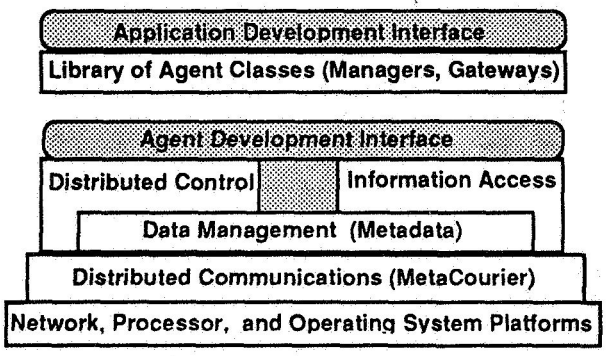

Figure .1: Architecture of the SOCIAL Toolset

Distributed Communication

SOCIAL's distributed computing utilities are organized in layers, enabling complex functions to be built up from simpler ones. The base or substrate layer of SOCIAL is the MetaCourier tool, which provides a high-level, modular distributed communications capability for passing information between applications based on heterogeneous languages, platforms, operating systems, networks, and network protocols [Sy90].

The Agent objects that integrate computer-based applications or resources are defined at SOCIAL's MetaCourier level. Developers use the MetaCourier API to pass messages between applications and their embedding Agents, as well as among application Agents. Messages typically consist of: commands that an Agent passes directly into its embedded application, such as database queries or calls to execute signal processing programs; data arguments to program commands that an Agent might call to invoke its embedded application; and symbolic flags or keywords that signal the Agent to invoke one or another fully preprogrammed interactions with its embedded application. For example, a high-level MetaCourier API call issued from a local LISP-based application Agent such as:

```
(Tell :agent 'sensor-monitor :sys 'Symb
'(poll measurement-Z))
```

transports the message contents, in this case a command to poll measurement-X, from the calling program to the Agent ***sensor-monitor*** resident on platform ***Symb1***. The Tell function initiates a message transaction based on an asynchronous communication model; the application Agent that issues such a message can immediately move on to other processing tasks. The MetaCourier API also provides a synchronous "Tell-and-Block" message function for "wait-and-see" processing models.

Agents contain two procedural methods that control the processing of messages, called in-filters and out-filters. In-filters parse incoming messages according to the argument list structure specified when the Agent is defined. After parsing the message, an in-filter typically either invokes the Agent's embedded resource or application, or passes the message (which it may modify) onto another Agent. The MetaCourier semantic model entails a directed acyclic computational graph of passed messages. When no further passes are required, the in-filter of the terminal Agent runs to completion. This Agent's out-filter method is then executed to prepare a message reply, which is automatically returned (and possibly modified) through the out-filters of intermediate Agents back to the originating Agent (i.e., the target Agent for the original Tell call).

A MetaCourier runtime kernel resides on each

application host. The kernel provides: (a) a uniform message-passing interface across network platforms; and (b) a scheduler for managing messages and Agent processes (i.e., executing filter methods). Each Agent contains two attributes that specify associated Host and an Environment objects. These MetaCourier objects define particular hardware and software execution contexts for Agents, including the host processor type, operating system, network type and address, language compiler, linker, and editor. The MetaCourier kernel uses the Host and Environment associations to manage the hardware and software platform specific dependencies that arise in transporting messages between heterogeneous, distributed Agents (cf. Figure .2).

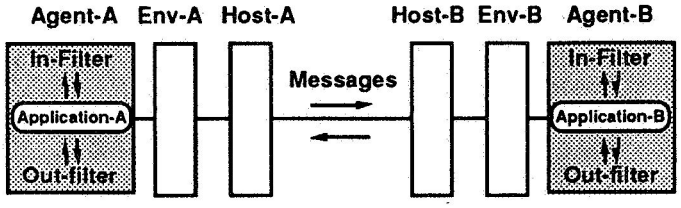

Figure .2: Operational Model of MetaCourier

MetaCourier's high-level message-based API is basically identical across different languages such as LISP or C. MetaCourier's communication model is also symmetrical or "peer-to-peer". In contrast, client-server models based on remote procedure call communication formalisms (RPCs) are asymmetric: only clients can initiate communication and while multiple clients can interact with a particular server, a specific client process can only interact with a particular server. Moreover, until recently, RPCs were restricted to inefficient synchronous (i.e., blocking) communications.

Data Specification and Translation

A major difficulty in getting heterogeneous applications and information resources to interact with one another is the basic incompatibility of their underlying models for representing data, knowledge, and commands. These problems are compounded when applications are distributed across heterogeneous computing platforms with different data architectures (e.g., opposing byte ordering conventions).

SOCIAL's Metadata subsystem addresses these complex compatibility problems. MetaData provides an object-oriented data model for specifying and manipulating data and data types across disparate applications and platforms. Developers access these tools through a dedicated API. MetaData handles three basic tasks: (a) encoding and decoding basic data types (e.g., character, integer, float), transparently across different machine architectures; (b) describing data of arbitrarily complex abstract types (e.g., database records, frames, b-trees), in a uniform object-oriented model; and (c) mapping across data models native to particular applications and SOCIAL's generic data model. Like MetaCourier, MetaData's object-oriented API is basically uniform across different programming languages. Also, MetaData allows new types to be defined and manipulated dynamically at runtime. Most alternative data management tools, such as XDR, are static and non-object-oriented.

SOCIAL integrates MetaData with MetaCourier to obtain transparent distributed communication of complex data and data types across heterogeneous computer platforms as well as across disparate applications: developers embed MetaData API function calls within the in-filter and out-filter methods of interacting Agents, using MetaCourier messages to transport MetaData objects across applications residing on distributed hosts. MetaData API functions decode and encode message contents, mapping information to and from the native representational models of source and target applications and MetaData objects. SOCIAL thereby separates distributed communication from data specification and translation, and cleanly partitions both kinds of generic functionality from application-specific processing.

Distributed Control and Information Access

SOCIAL's third layer of object-oriented tools establishes custom, high-level API interfaces for Agent classes specialized for particular integration or coordination functionality. Lower-level MetaCourier

and MetaData API functions are used to construct these Agent API data and control interfaces. The high-level APIs largely conceal MetaCourier and MetaData interfaces from SOCIAL users. Thus, developers of distributed systems typically use the predefined specialized Agent classes as the top-level building blocks for satisfying their particular architectural requirements, accessing the functionality of each such Agent type through the dedicated high-level API interfaces.

For example, applications and data stores are often constructed using standard development tools such as database management systems (DBMSs) and AI shells. SOCIAL Database and Knowledge ***Gateway*** Agent classes abstract and isolate the application-independent aspects of the control and data interfaces to such shells as generic, reusable API interfaces. Similarly, ***Managers*** are specialized Agents for coordinating applications integrated via Gateways and other kinds of SOCIAL Agents to work together cooperatively. The HDC-Manager, described in the following sections, defines one kind of centralized model for distributed control of application Agents. If necessary, developers can access SOCIAL's various tool layers to modify or extend existing Agent APIs and define corresponding new subclasses of Gateway, Manager, or fully custom Agents.

Gateway and Manager Agent APIs depend heavily on MetaCourier and MetaData capabilities for controlling and manipulating messages. For instance, the root Knowledge Gateway Agent class defines standard MetaCourier in-filter and out-filter methods. The generic in-filter parses and processes messages that represent either: (a) incoming requests initiated by other application Agents; or (b) requests initiated by the Agent's embedded application to pass on to other application Agents. Similarly, the generic out-filter either: (a) assembles responses to in-coming requests; or (b) relays replies to prior out-going requests back to the embedded application. All subclasses of Knowledge Gateway Agents inherit these generic shell- and application-independent control behaviors.

SOCIAL uses MetaData to define a uniform representational model for data and knowledge structures such as relational tuples, facts, fact-groups, rules, and frames. Subclasses of the root Knowledge Gateway Agent class define API functions that map transparently between SOCIAL's canonical representation and the knowledge models native to particular AI development tools. For example, CLIPS Knowledge Gateway Agents translate between MetaData frames and CLIPS deftemplates, while KEE Gateway Agents map between MetaData frames and KEE unit objects. MetaData objects are also used to pass commands to AI shells associated with Gateway Agent subclasses (e.g., to reset a fact base, execute a rule-based inference engine, or load application files). This uniform mapping approach simplifies the problem of interconnecting N disparate systems from O(N*N) to O(N).

Developers integrate a shell-based application by embedding it in an instance of the relevant Gateway Agent subclass, using its shell-specific API functions to program the required interactions: the API calls specify the particular data or commands to be injected into, or extracted from, the embedded application's knowledge bases through the shell's control interfaces. The strategy of modular, specialized interface functionality used in SOCIAL's predefined Gateway Agent classes is directly applicable for designing custom types of Agent for integrating standalone applications or systems implemented using in-house, proprietary development shells.

THE HDC-MANAGER AGENT

Taken together, the SOCIAL tools described thus far - MetaCourier, MetaData, and Gateway Agents - provide adequate support for integrating conventional heterogeneous software systems. The interactions in such systems are relatively few in number and can be prescribed in a fixed, determinate order. For example, satellite telemetry data can be fed to Agents embedding signal processing and pattern recognition programs to clean and filter data and check for significant events. Database Gateway Agents would be used to dump all data to

archival storage, while extracting and writing noteworthy events to on-line storage systems. Scientists might then study the data through Agents that embed data analysis and visualization programs, possibly through an intermediary User Interface Agent. The developer of such a system would embed the constituent applications and databases in suitable Agents and specify their direct, one-to-one interactions in terms of the relevant Agent APIs.

Once autonomous knowlege-based systems are incorporated into a distributed system, integration tools alone are no longer fully adequate. First, the sequencing or composition of behaviors both within and across autonomous systems is typically determined dynamically, based on the content of incoming data at run-time. A decentralized approach to managing such data-driven behaviors quickly becomes intractable. The problem is particularly acute in distributed systems that evolve over an extended lifecycle, in which application elements are enhanced, added, superseded, or reorganized (i.e., broken apart or consolidated), over time.

Second, complex organizational relationships emerge among clusters of applications in large-scale distributed intelligent systems. For example, programs that automate on-line operations of computer networks and other complex systems are naturally coupled more closely to one another than to decision or maintenance support tools for the same target domain. At the same time, interactions regularly take place between applications across functional groupings. For instance, planning and scheduling tools (decision support) dictate system configuration activities (operations support), while behavioral anomalies detected in on-line systems (operations support) trigger and guide troubleshooting, diagnosis, and repair activities (maintenance support). Ideally, interfaces for such cross-group interactions should be designed at the cluster level, to minimize sensitivity to modifications within functional groups.

In short, the intelligent application elements of complex distributed systems must not only be integrated, but also ***coordinated***. Systematic coordination is necessary both to manage large numbers of dynamically evolving interaction pathways and to capture complex logical relationships among functional elements. The HDC-Manager is the first specialized class of SOCIAL Agents developed to address these organizational requirements.

HDC-Manager Functionality

In essence, the HDC model performs the same role in a complex distributed system played by a human manager in a large organization. The HDC-Manager is associated with one or more application Agents, called its ***subordinates***. The HDC-Manager's specific functions or responsibilities with respect to these other Agents include:

- providing a centralized Index knowledge base that specifies: each available information source or problem-solving service; the subordinate Agent that can provide that resource; the Agent's logical address; and a procedure for converting data in a generic resource request into a message format that is suitable for that server Agent;
- analyzing tasks requesting information or problem-solving services based on the Index knowledge base and routing suitable messages to the relevant subordinate Agents to accomplish those tasks;
- mediating all interactions with external (i.e., non-subordinate) Agents;
- providing a centralized Bulletin-Board to store problem-solving results and other data of common utility for shared access by subordinate and external Agents;

The HDC-Manager realizes the advantages of a centralized control architecture in a complex distributed system:

- efficient *global* coordination of individual problem-solving Agents;
- modularity;

- extensibility and maintainability;
- support for heterogeneity.

Modularity derives from the HDC-Manager's centralized Bulletin-Board and Index knowledge base. Each Index entry identifies services available from subordinate Agents symbolically (e.g., find-fault-precedents). Each Bulletin-Board posting identifies the item type, such as service request or reply, the posting Agent, and the requesting Agent (when appropriate). Subordinate Agents do not need to know about the functionality, structure, or even the existence of any other application Agents; they only require: (a) the generic high-level API interface to the HDC-Manager Agent; and (b) knowledge of the symbolic names used by the HDC-Manager to index the resource types available within a specific distributed application. The same minimal requirements hold for external application and Manager Agents that need to interact with an HDC-Manager to obtain information or services from its subordinates.

Extensibility and maintainability follow from the HDC-Manager's modular architecture: new subordinate Agents are incorporated simply by: (a) updating the Index knowledge base with appropriate entries to describe its services; and (b) extending other subordinate Agents, as needed, to be able to request new capabilities and process the results. Moreover, existing subordinate Agents can be reconfigured with minimal disruption. For example, suppose that problem-solving functions in one subordinate Agent are reallocated to some other application Agent, old or new. Neither the Manager's other subordinates nor any external Agents have to be modified; only the HDC-Manager Index knowledge base needs to be updated to reflect the configuration changes.

Heterogeneity follows from the modular nature of SOCIAL Agents in general. The high-level API to the HDC-Manager Agent's coordination capabilities is distinct from, but fully compatible with, the API interfaces used to embed applications with Gateways or other kinds of SOCIAL Agents. Thus, the HDC-Manager Agent operates transparently with respect to the physical distribution and internal architectures (i.e., communication, control, and knowledge structures), of subordinate and external Agents with which it interacts. Consequently, an HDC-Manager can subordinate standalone application Agents, Gateway Agents, and other Manager Agents, HDC or otherwise, with equal ease. In particular, HDC-Manager Agents can be organized in a nested hierarchy to support large complex distributed systems, as illustrated in Figure .3.

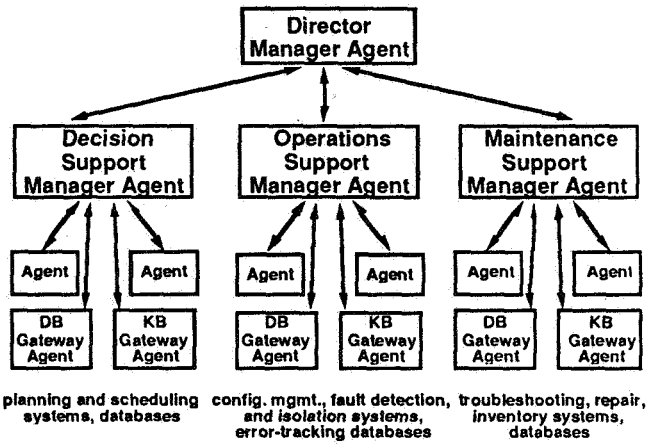

Figure .3: Hierarchy of HDC-Manager Agents

HDC-Manager Architecture and Operation

The HDC-Manager Agent was implemented using SOCIAL and Common Lisp in a uniform, fully object-oriented manner. The Agent is comprised of a collection of state variables and utility methods (cf. Figure .4). The value for each state variable consists of a list of MetaData objects, such as Agent-Index-Items. Each such object type has associated test predicate, instance creation, and access functions (e.g., Agent-Index-ItemP, Create-Agent-Index-Item, Check-Agent-Index). These low-level functions, written using the MetaData API, are invoked through a higher-level HDC-Manager API and are transparent to developers. Five state variables store the static and dynamic information necessary for the HDC-Manager to function:

- a Task-Agenda for posting, prioritizing, and dispatching service request Tasks;
- a Bulletin-Board for posting Task results and other data items to a common memory store accessible to all subordinate and external Agents;
- an Index knowledge base that enumerates subordinate Agents, their logical locations, server capabilities, and functions for assembling data into suitable command message formats;
- a set of Prioritization Conditions for ordering the Agenda queue of pending Tasks to be routed by the Manager;
- an Activity Log for tracing and debugging Manager behavior during application development;

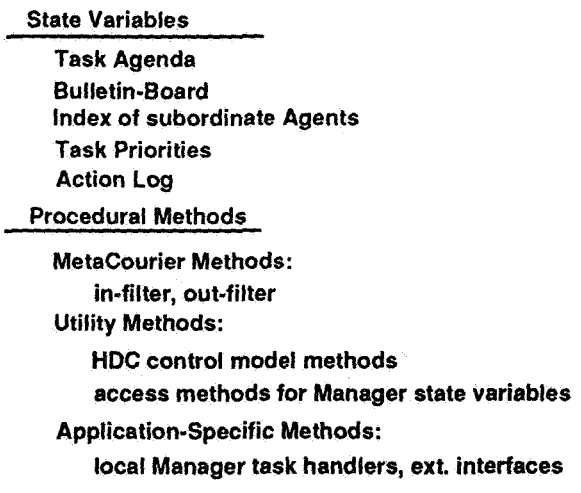

Figure .4: HDC-Manager Agent Structures

Currently, the Index knowledge base and the Prioritization conditions represent static structures that are specified once, when an HDC-Manager Agent is first defined to coordinate a specific set of applications. Typically, changes are made during development, but infrequently thereafter, when subordinate application Agents are added, removed, or restructured. (Self-adapting systems could modify Priority Conditions or even create new server Agents dynamically; however, we have not yet investigated such possibilities.) The remaining three state variables are more dynamic structures: their contents change continually as the HDC-Manager regulates an operational distributed system.

The HDC-Manager Agent also incorporates four sets of supporting procedural methods:

- in-filter and out-filter methods for parsing and responding to incoming MetaCourier messages and processing replies to outgoing messages;
- auxiliary API utility methods that realize the HDC-Manager's global hierarchical control model;
- auxiliary API utility methods for creating and modifying HDC-Manager data items, posting them to and retrieving them from the Agent's state variables;
- optional methods for handling application Tasks within the HDC-Manager itself. Such methods may be called for when a Manager Agent is configured as a subordinate to other Manager Agents.

The HDC-Manager's specialized coordination functions and information structures are accessed through a dedicated API (cf. Figure .5). This API is implemented via lower-level MetaData and MetaCourier capabilities and APIs. The high-level API shields SOCIAL developers from the underlying mechanics of packaging, transporting, and deciphering messages containing HDC-Manager commands and data structures among heterogeneous Agents. The API utility methods for accessing HDC-Manager state variables and their contents include:

- two methods for searching the Index Knowledge Base and Bulletin-Board. Both methods call a generic symbolic pattern-matching function with application-specific search conditions;
- a single Create-Item method, which dispatches to the various functions that create instances

of HDC-Manager MetaData object types: Index and Bulletin-Board entry items; Priority-Conditions, and Tasks (Service Requests);

- four parallel methods for modifying HDC-Manager state variables: Task-Agenda, Index Knowledge Base, Bulletin-Board, and Priority-Conditions; Each method supports keyword options for resetting the variables, posting and deleting specific data items;

The HDC-Manager API is also used to invoke the utility methods that implement the HDC control model. These methods are generally triggered automatically, from the Manager's predefined in-filter and out-filter, but can be activated by other Agents as required. HDC-Manager control methods include:

- a Command-Manager method, which dispatches API command messages, either to internal HDC-Manager API methods *or* to the Command-Manager of another HDC-Manager Agent. This method also traps illegal commands;
- an Initialize method for resetting the HDC-Manager state variables and performing any application-specific actions;
- a Prioritize-Agenda method for sorting pending Tasks for the Manager to dispatch in accordance with the declarative ordering conditions specified in the Priorities-Conditions state variable. Each condition specifies a Task Attribute (e.g., service-category, priority), and an optional list of Attribute values for ordinal sorting;
- a Task-Dispatcher method, which uses the Index Knowledge Base to generate and send a suitable command message to the relevant subordinate Agent requesting the service specified in a Task;
- a top-level Control-Cycle method that invokes the Prioritize-Agenda and Task-Dispatcher methods that ground the HDC model;
- a Log-Utility method. A menu-driven trace/debug facility can be used to toggle the Activity Log, which tracks all messages processed by the Command-Manager, and other flags that trace of all runtime modifications to Manager state variables.

Control Methods	Data Structure Accessors
Command-Manager	Check-Agent-Index
Control-Cycle	Check-Bulletin-Board
Initialize	Create-Item
Log-Utility	Modify-Agenda
Prioritize-Agenda	Modify-Agent-Index
Task-Dispatcher	Modify-Bulletin-Board
	Modify-Priority-Conditions

Figure .5: HDC-Manager Agent Utility Methods

The Command-Manager enforces regulated hierarchical control channels. A subordinate Agent can communicate with any HDC-Manager Agent within a Manager hierarchy; however, any such message is processed first by the Agent's immediate superior and then by all intervening Managers. This design makes it possible to define and enforce alternative organizational reporting policies. The default policy is very flexible: messages are tracked, but passed along the Manager hierarchy without filtering. Thus, any subordinate Agent can access the Bulletin-Board or request services from other Managers through its immediate Manager. More or less restrictive control architectures may be appropriate under different conditions. For example, messages from subordinates to higher-level Managers in time-critical applications could be screened or prioritized. Parallel, antagonistic or competitive models can also be explored. Finally, the HDC-Manager architecture does not preclude a subordinate reporting to multiple Managers, thus permitting non-hierarchical models (e.g, matrix structures), or elaborate hybrid organizational structures.

The HDC-Manager was designed in a modular fashion to facilitate maintenance and extension. API accessor utility methods conform to uni-

form conventions for naming, argument call structures, and parallel behavior. For example, Tasks or other data structures can be modified by adding attributes (e.g., timestamps for First-In-First-Out ordering), or changing attribute names. The developer merely changes the relevant Create-X function (and possibly one of the API Check methods). New HDC-Manager state variables and data objects to populate them can be added by implementing appropriate MetaData type-checking predicates, creation and accessor functions, adding a case to the Create-Item API method, and extending the table that drives the Command-Manager and Task-Dispatcher control methods.

Similarly, the HDC-Manager API control methods share parallel argument call structures and can be modified or extended selectively. For example, the Initialize method can be customized to perform any actions required to load and initiate all subordinate Agents and their embedded applications. Moreover, as noted above, alternative organizational policies can be implemented to capture logical relationships specific to particular distributed systems. All such modifications are implementing by creating HDC-Manager Agent subclasses with custom methods that override the standard methods defined by the root Agent class. For instance, the default initialization behavior can be redefined simply by creating a subclass Agent with a new Initialize method that calls a Custom-Initialize function. Specialization preserves the structure of the original HDC-Manager Agent class for use in applications where it is suitable. At the same time, inheritance and functional abstraction (through method dispatching) promotes adaptability and compact definitions for customized Manager Agent subclasses.

USING THE HDC-MANAGER AGENT FOR LAUNCH OPERATIONS SUPPORT

Over the past decade, NASA Kennedy Space Center (KSC) has developed knowledge-based systems to increase automation of operations support tasks for the Space Shuttle fleet. Major applications include, operations support of the Shuttle Launch Processing System, monitoring, control, fault isolation and management of on-board Shuttle systems and Ground Support Equipment. Prototypes have been tested successfully (off-line) in support of several Shuttle missions and are currently being extended and refined for formal field testing and validation. Final deployment will require integrating these applications, both with one another and with existing Shuttle operations support systems.

KSC recently initiated the EXODUS project (Expert Systems for Operations Distributed Users) to prepare for this challenging systems integration task. As part of this effort, KSC is funding Symbiotics, Inc. to develop the SOCIAL toolset to help validate, refine, and ultimately implement the proposed EXODUS architecture [Ad90a]. Proof-of-concept prototypes have been constructed to demonstrate central EXODUS design concepts: distributed data transfer; non-intrusive physical distribution of knowledge bases from existing intelligent system to facilitate resource control and sharing; and integration of expert systems and databases via Gateway Agents for CLIPS, KEE, and Oracle development tools. This section describes a fourth EXODUS prototype, which used a SOCIAL HDC-Manager Agent to coordinate the fault isolation activities of two previously standalone intelligent systems.

Background on KSC Launch Operations

Processing, testing, and launching of Shuttle vehicles takes place at facilities dispersed across the KSC complex. Many activities, such as storing and loading fuels and controlling the environments of Shuttles on Launch Pads require elaborate electromechanical Ground Support Equipment. The Launch Processing System (LPS) supports all Shuttle preparation and test activities from arrival at KSC through to launch. The LPS provides the sole direct real-time interface between Shuttle engineers, Orbiter vehicles and payloads, and associated Ground Support Equipment [He87].

The locus of control for the LPS is the Firing Room, an integrated network of computers,

software, displays, controls, switches, data links and hardware interface devices. Firing Room computers are configured to perform independent LPS functions through application software loads. Shuttle engineers use computers configured as Consoles to remotely monitor and control specific vehicle and Ground Support systems. Each such application Console communicates with an associated Front-End Processor (FEP) computer that issues commands, polls sensors, and preprocesses sensor measurement data to detect significant changes and exceptional values. These computers are connected to data busses and telemetry channels that interface with Shuttles and Ground Support Equipment.

The LPS Operations team ensures that KSC's four independent Firing Rooms are available continuously, in appropriate error-free configurations, to support test requirements such as Launch Countdown or Orbiter Power-up sequences for the Shuttle fleet. A dedicated Console computer is configured for these Operations support functions in each Firing Room. This computer displays messages triggered by the LPS Operating system that signal anomalous events such as improper register values or expiring process timers. The Operations Console supports other conventional programs for monitoring and retrieving Firing Room status data as well.

OPERA (for Operations Analyst) consists of an integrated collection of expert systems that automates many critical LPS Operations support functions [Ad89b]. OPERA taps into the same data stream of error messages that the LPS sends to the Operations Console. OPERA's primary expert systems monitor the data stream for anomalies and assist LPS Operations users in isolating and managing faults by recommending troubleshooting, recovery and/or workaround procedures. In effect, OPERA *retrofits* the Operations Console with knowledge-based fault isolation capabilities. The system is implemented in KEE on Texas Instruments Lisp Machines.

GPC-X is a prototype expert system for isolating faults in the Shuttle vehicle's on-board computer systems, or GPCs. GPC-X monitors (simulated) PCM telemetry data to detect and isolate faults in communications between Shuttle GPC computers and their associated GPC-FEP computers in LPS Firing Rooms. The GPC-X prototype is implemented in CLIPS on a Sun Workstation.

Coordinating Fault Diagnosis with HDC-Managers

One type of memory hardware fault in GPC computers manifests itself during switchovers of Launch Data Buses. These buses connect GPCs to GPC-FEPs until just prior to launch, when communications are transferred to telemetry links. Unfortunately, the data stream available to GPC-X does not provide any visibility into the occurrence of Launch Data Bus switchovers. Thus, GPC-X can propose, but not test certain fault hypotheses about GPC problems. However, switchover events are monitored by the LPS Operating System, which triggers messages that can be detected by OPERA.

Typical of the current generation of knowledge-based systems, OPERA and GPC-X were developed independently of one another, using different representation schemes, reasoning and control models, software and hardware platforms. More critically, neither system possesses internal capabilities for modeling or communicating with (remote) peer systems. The EXODUS prototype demonstrates the use of SOCIAL Agents to rectify these shortcomings (cf. Figure .6). The distributed application uses two Knowledge Gateway Agents to integrate OPERA and GPC-X. An HDC-Manager Agent mediates interactions between the OPERA and GPC-X Agents, coordinating their independent fault isolation activities obtain enhanced diagnostic results.

Specifically, GPC-X, at the appropriate point in its rule-based fault isolation activities, issues a request via its Gateway Agent to check for Launch Data Bus switchovers to the HDC-Manager. The request is triggered by adding a simple consequent clause of the form (GW-Return LDB-Switchover-Check) to the CLIPS rule that proposes the memory fault hypothesis. GW-Return is a custom

C function defined in the SOCIAL API interface for embedding CLIPS. When the rule fires, GW-Return interacts with the GPC-X CLIPS Gateway Agent, causing it to formulate a message to the HDC-Manager containing a Modify-Agenda request to add a MetaData Task object for the LDB-Switchover-Check service. The HDC-Manager's Command-Manager dispatches this request, causing the Task to be posted to the Task-Agenda.

Next, the HDC-Manager executes the Control-Cycle method, which results in the Task being processed via the Task Dispatcher. First, the Index knowledge base is searched for a server Agent for LDB-Switchover-Checks. The search identifies the OPERA Gateway Agent as a suitable server. Task data is then reformulated into a command message using the procedure specified by the Index Knowledge Base. The message is then passed to the OPERA Gateway Agent, whose in-filter method performs a search of the knowledge base used by OPERA to store and interpret LPS Operating System error messages. The objective is to locate error messages (represented as KEE units) indicative of LDB Switchover events. Search results are encoded within a Manager Bulletin-Board MetaData object, which the OPERA Gateway's out-filter method returns as a message containing an API command to post that object to the Manager's Bulletin-Board. In this prototype, the OPERA Gateway contains *all* of the task processing logic: OPERA itself is a passive participant that continues its monitoring and fault isolation activities without significant interruption.

The GPC-X CLIPS Gateway Agent queries the HDC-Manager to check the Bulletin-Board for a response to its LDB-Switchover-Check request and retrieves the results. The retrieved Bulletin-Board item is decoded and the answer is converted into a fact that is asserted into the GPC-X fact base. Finally, the Gateway activates the CLIPS rule engine to complete GPC fault diagnosis. Obviously, new rules have to be added to GPC-X to exploit the newly available hypothesis test data. However, all of the basic integration and coordination logic is supplied by the embedding GPC-X Gateway Agent or the HDC-Manager.

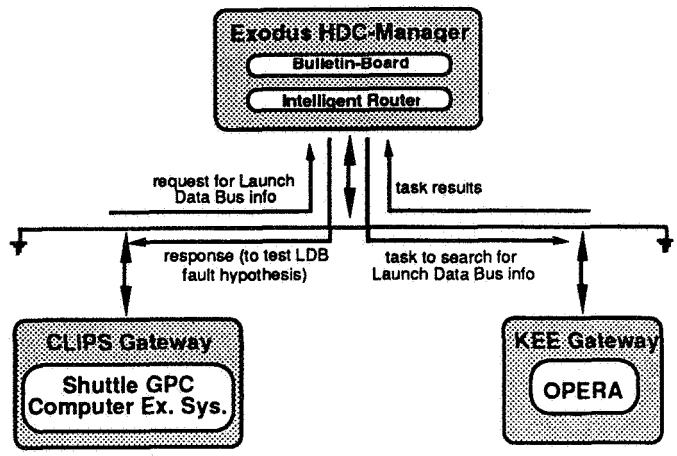

Figure .6: Hierarchical Coordination in EXODUS

This EXODUS prototype illustrates non-intrusive system-level coordination of distributed applications that solve problems at the subsystem level of Shuttle Operations: neither OPERA nor GPC-X are capable of accomplishing the task of confirming or rejecting the GPC memory fault hypothesis individually. GPC-X generates fault candidates, but lacks sufficient resources to complete diagnosis, which requires ***both*** generate and test capabilities. OPERA automatically detect LPS error messages that are relevant to GPC-X's fault test requirements. However it lacks contextual knowledge about GPC computers - their architecture, behavior, fault modes and symptoms - to recognize the significance of such data. OPERA also lacks the capabilities to communicate its interpretations of LPS data back to GPC-X to complete diagnosis.

Together with the Gateway application Agents, the HDC-Manager provides the links required to combine and utilize the otherwise isolated or fragmented knowledge about Shuttle and Firing Room systems and their relationships to one another. The resulting coordination architecture is non-intrusive in that neither application was modified to include direct knowledge of the other system, its interfaces, knowledge model, or delivery platform. The HDC-Manager introduces an isolating layer of abstraction; application Agents need only know how to communicate with the HDC-Manager to request services and retrieve responses for their embedded

applications.

The proposed design for the complete EXODUS system specifies an extended HDC model to integrate and coordinate all of KSC's operations support applications across areas of functional overlap. A SOCIAL HDC-Manager Agent will support architectural changes as EXODUS evolves through its lifecycle of initial deployment, maintenance, and enhancement. Initial EXODUS applications are loosely-coupled and will interact relatively infrequently. Consequently, the HDC-Manager's centralized control strategy does not entail serious performance penalties. As new applications are added and interaction traffic increases, bottlenecks will be addressed by reorganizing the EXODUS control architecture in terms of hierarchies of HDC-Manager Agents, much as growing human organizations evolve.

RELATED WORK

The HDC-Manager generalizes and extends a hierarchical distributed control (HDC) model originally developed for NASA's Operations Analyst (OPERA) system [Ad89a]. The OPERA version of the control model was implemented using distributed blackboard objects [Ad89b]. OPERA requires all applications to be co-resident and to be implemented using KEE, and only supports a single Manager and subordinate group. The extended HDC model relaxes these restrictions using MetaCourier, MetaData, API tools, SOCIAL Agents.

Most research in Distributed Artificial Intelligence (DAI) has focused on domains involving a single complex problem, such as data fusion. Control schemes have emphasized purely local coordination methods to achieve cooperation among intelligent systems [Bo88,Hu87]. For example, [Le83] employs a homogeneous collection of blackboards that interpret data from a (simulated) network of spatially distributed sensors to reconstruct vehicle positions and movements. Data and hypotheses are shared across adjacent sensor regions. Solutions emerge consensually, without global management. Decentralized control entails significant performance overhead from duplicated processing. As argued earlier, localized coordination strategies can be cumbersome and difficult to maintain for heterogeneous, evolving "multiple problem" DAI systems.

The MACE DAI tool [Ga86] incorporates manager agents for centralized routing of messages among agents, closely resembling the organizing role played by SOCIAL HDC-Managers. However, it is not clear that MACE managers can be configured in multi-level hierarchies. Moreover, MACE managers do not provide shared memory Bulletin-Boards. MACE and several other distributed system tools such as ABE [Ha88] and CRONUS [Sh86] insulate developers from low level distributed computing functions and support message sending among processes across computer networks. However, unlike SOCIAL, these tools do not implement generic distributed services in uniformly object-oriented layered modules that are accessible to developers for customizing. In addition, SOCIAL provides more extensive tools for integrating across languages and software development shells.

FUTURE WORK

Future development will extend SOCIAL's library of Manager Agents. Alternative organizational models will explore alternative types of nonhierarchical cooperative coupling. For example, we are investigating control behaviors based on group-based tasking [Br89], as one approach to providing fault tolerance in distributed systems: groups can be used to define sets of application Agents that duplicate support for given services. A group Agent that could detect loss of an Agent configured to provide a service (e.g., due to dropped network links or host platform failures), could activate another member Agent to resume the service. Redundancy and recoverability are critical prerequisites for distributed systems in mission-critical space and military applications. We also plan to implement C versions of SOCIAL Manager and Gateway Agents that are currently Lisp-based.

CONCLUSIONS

Development tools for distributed intelligent systems must be modular and non-intrusive to: (a) facilitate integration of existing, standalone systems "after the fact;" and (b) minimize lifecycle costs for maintaining, enhancing, and re-verifying systems. Tools for building distributed systems must be able to coordinate as well as integrate autonomous application elements. Coordination is necessary to manage large numbers of dynamic interaction pathways and to capture complex organizational relationships among application elements. SOCIAL provides a unified set of object-oriented tools that address all of these requirements. The HDC-Manager Agent realizes a hierarchical distributed control model that adopts a highly centralized approach to coordination. Developers use a high-level Application Programming Interface to access the HDC-Manager's coordination capabilities. The API conceals lower-level SOCIAL tools for transparent distributed communication, control, and data management.

SOCIAL has broad applicability for distributed intelligent systems that are being developed in space-related domains. Knowledge-based and conventional tools for managing and analyzing data need to be coupled to help space scientists explore and utilize NASA's growing stores of astronomical and environmental information. Linking short- and long-term scheduling and planning tools will improve decision support capabilities for complex space missions. EXODUS-like architectures can increase automation of operations support by coordinating autonomous tools across subsystems and functional areas (e.g., configuration, anomaly detection, diagnosis and correction). Example domains include payload and Shuttle processing, computer and communications networks, and vehicle or Space Station subsystems (e.g., power generation, power distribution, mission payloads, life support). Finally, flight and mission control centers can enhance automation and safety in directing launches, satellites, and space probes, by combining decision and operations support tools into fully unified, cooperating systems.

Acknowledgments

SOCIAL was designed and implemented by the author, Bruce Cottman and Rick Wood. Development of SOCIAL has been sponsored by NASA Kennedy Space Center under contract NAS10-11606. Astrid Heard of the Advanced Projects Office at Kennedy Space Center has provided invaluable support and suggestions for the SOCIAL effort. Ms. Heard also initiated and directs the EXODUS project. Meta-Courier was developed by Robert Paslay, Bruce Nilo, and Robert Silva, with funding support from the U.S. Army Signals Warfare Center under Contract DAAB10-87-C-0053.

REFERENCES

[Ad90a] Adler, R.M. and Cottman, B.H. "EXODUS: Integrating Intelligent Systems for Launch Operations Support." *Proceedings, Space Operations, Applications, and Research Symposium (SOAR90).* Albuquerque, NM. June 26-28, 1990.

[Ad90b] Adler, R.M. and Cottman, B.H. "A Development Framework for AI Based Distributed Operations Support Systems." *Proceedings Fifth Conference on AI for Space Applications.* Huntsville, AL, May 21-23, 1990.

[Ad89a] Adler, R.M., Heard, A., and Hosken, R.B. "OPERA - An Expert Operations Analyst for A Distributed Computer Network." *Proceedings Annual AI Systems in Government Conference, Computer Society of the IEEE.* Washington, D.C., March 27-31, 1989.

[Ad89b] Adler, R.M. "A Distributed Blackboard Architecture for Integrating Loosely-Coupled Knowledge-Based Systems." *Intelligent Systems Review.* 1, 4, Summer, 1989, Association for Intelligent Systems Technology, E. Syracuse, NY.

[Br89] K. Birman et. al. *The ISIS System Manual V1.2.* Department of Computer Science, Cornell University, Ithaca, NY, June 1989.

[Bo88] Bond, A.H., and Gasser, L. (eds.) *Readings in Distributed Artificial Intelligence.* Morgan-Kaufmann, San Mateo, CA, 1988.

[Ga86] Gasser, L., Braganza, C., and Herman, N. *MACE: A Flexible Testbed for Distributed AI Research.* Distributed Artificial Intelligence Group, Computer Sci. Dept. USC, 9-Aug-1986.

[Ha88] Hayes-Roth, F., Erman, L.D., Fouse, S., Lark, J.S., and Davidson, J. (1988). "ABE: A Cooperative Operating System and Development Environment." in A.H. Bond and L. Gasser, (eds.) *Readings in Distributed Artificial Intelligence.* Morgan-Kaufmann, San Mateo, CA, 1988.

[Hu87] Huhns, M.N. (ed.) *Distributed Artificial Intelligence.* Morgan-Kaufmann, Los Altos, California, 1987.

[Le83] Lesser, V.R and Corkill, D.D. "The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks." *AI Magazine.* Fall 1983 pp. 15-33.

[Sh86] Schantz, R., Thomas, R. and Bono, G. "The Architecture of the Cronus Distributed Operating System. *Proceedings 6th International Conference on Distributed Computing Systems.* May, 1986.

[Sy90] Symbiotics, Inc. *Object-Oriented Heterogeneous Distributed Computing with MetaCourier.* Technical Report, Cambridge, MA, March, 1990.