Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

2000

The Bond Agent System and Applications

Ladislau Boloni

Kyungkoo Jun

Krzysztof Palacz

Radu Sion

Dan C. Marinescu

Report Number:
00-008

Boloni, Ladislau; Jun, Kyungkoo; Palacz, Krzysztof; Sion, Radu; and Marinescu, Dan C., "The Bond Agent System and Applications" (2000). *Department of Computer Science Technical Reports.* Paper 1486.
https://docs.lib.purdue.edu/cstech/1486

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

THE BOND AGENT SYSTEM AND APPLICATIONS

Ladislau Boloni
Kyungkoo Jun
Krzystof Palacz
Radu Sion
Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #00-008
March 2000

The Bond Agent System and Applications

Ladislau Bölöni, Kyungkoo Jun, Krzysztof Palacz, Radu Sion, and Dan C. Marinescu
Computer Sciences Department, Purdue University
West Lafayette, IN, 47907, USA
Email: [boloni, junkk, palacz, sion, dcm]@cs.purdue.edu

March 2, 2000

Abstract

In this paper we present the basic design philosophy of the Bond agent system, the multi-plane agent model and the component-based architecture implementing the model. We discuss several applications of Bond agents: resource discovery, an adaptive video service, a workflow management system, a system of agents for remote monitoring of web servers, and a network of PDE solvers.

1 Introduction

We present some of the features of the Bond agent system and several applications of it. The original goal was to create an infrastructure for a Virtual Laboratory and to support scheduling of complex tasks and data annotation for data intensive applications. The Virtual Laboratory is expected to facilitate remote control and monitoring of experiments, data analysis, replaying of a past experiment, knowledge sharing among scientists scattered around the country. Early on, we realized that, due to the complexity of the tasks involved, the infrastructure should support knowledge and workflow management and be based upon a distributed object system. We created an agent model and a component-based architecture to assemble the agents [9].

Our thinking and design choices were influenced by existing systems and, whenever possible, we adopted ideas and integrated implementations fitting our agent model. We integrated with relative ease JESS, a Java Expert System Shell from Sandia National Laboratory, [20] and we are in the process of designing a planning engine and integrating a knowledge management system, see Section 3.1.

Now we outline our views regarding several controversial issues in the area of software agents and present our design choices. The intelligent agents and the distributed objects and systems communities have slightly different views regarding the future of software agents. The first group is primarily concerned with intelligent agents and applications where agents are indispensable, e.g. space exploration or robotics, where advanced planning and unrestricted autonomy are necessary. The second group believes in agents with a wide range of intelligence and autonomy capabilities, useful for the development of the next generation Internet-based applications.

A recent paper by Nwana and Ndumu [30] provides a lucid but somber analysis of the field. The authors review the promises and evaluate the progress of the last few years in the field and conclude that, while progress has been made in several areas including information discovery, ontologies, agent communication, reasoning and coordination, monitoring and integration of agents and legacy software for the past five years, the progress in the software agents field has been by and large, slow and marred by recycling of concepts developed earlier.

The stagnation of the field is attributed by the authors of the study to several causes: (a) lack of focus on problems that indeed require agent technology as opposed to problems that can be solved by traditional distributed system. (b) inability to identify a "killer application", and (c) a premature tendency towards formalization, attributable to many academic agent researchers that focus on manufactured agent applications rather than on realistic ones.

Our view regarding applications of software agents is different, we see a fair number of applications where software agents technology may have a significant impact though more traditional approaches are possible. The design of complex systems requires components with different degrees of autonomy, intelligence and mobility, any component may or may not be viewed as an agent depending upon the particular circumstances it is used. Agents could be used to support: (a) access to services from platforms ranging from supercomputers to hand-held devices, (b) composition and customization of services, (c) information discovery, (d) resource management, (e) negotiations and so on.

At the same time, we regard the software engineering of agents as a major concern and believe that software agent technology should be integrated with other methods and technologies used to build complex open systems, including object-oriented technology, concurrency, distribution. If we have to use special agent communication and content languages, design our own societal services, use special toolkits for building agents, while waiting for someone to discover a "killer application", it is very likely that, after five years from now, an equally somber review of the field could be expected.

Another controversial aspect of agents is related to mobility. Code mobility is a long-term obsession of distributed system designers, has its roots in work done last decade at MIT on remote evaluation, on process migration research, fashionable two decades ago, and can be related to the Xerox Worm. Some question the usefulness of agent mobility, others point out the tremendous challenges posed by it, security being often at the top of the list, and argue that we should address first the very difficult problems posed by immobile agents [29]. Systems like Voyager [21] and IBM's Aglets [28] were specifically designed to support agent mobility while others like Retsina [34] or Zeus [31] ignore it. Systems like AgentTCL [27] and Bond support agent mobility.

We believe that there are applications where support for agent mobility is critical, the agent is an integral part of the model, e.g. in active networks [8] [39] [36]. There are applications where mobile agents may have a significant advantage in terms of either functionality or performance compared with more traditional techniques: (a) extensible servers, applications where a user is represented by an agent installed at a remote server location, (b) data-intensive applications where it is impractical to move data, (c) applications in mobile computing area where the user is intermittently connected to the network and resources available on the network access device are insufficient, (d) dynamic deployment of software [29].

The paper is organized as follows. Section 2 and 3 present defining features of the Bond system and work in progress. Several applications of Bond agents are examined in Section 4.

2 The Bond Agent System

2.1 The Agent Model

The Bond agent model is influenced by the definition of an agent given by Stan Franklin and Art Graesser [19]: *an autonomous agent is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.*

The agent execution model assumes that the agent has an explicit goal. Agents receive external events and generate actions. However, the actions of an agent are determined by the pursue of its goal. Of course, events may trigger immediate responses, but agents perform actions even without any external input. An agent terminates its execution when its goal is accomplished.

Since an agent can emulate every other execution model, some researchers are inclined to view every program as an agent. Although this approach may be sometimes useful, it does not lend itself to efficient implementations, the simpler the execution model the more optimal implementation is possible. For example, we can optimize the response time of a stateless server far better than of an agent with a complex state.

The structure of Bond agents reflects this execution model and allows an agent to interact with other agents, take actions on its own, and mainain a complex state. Although this framework can be used to implement non-interactive programs or servers in the classical sense, these implementations may be suboptimal.

Bond is based upon the AM_1^{mp} model [10]. We believe this model to be well suited to an object-oriented implementation, though less powerful than BDI. This model allows us to reason about agents while using an object-oriented programming style.

An important aspect of agent design is the communication style. Most agent systems use a message-oriented style although recently agents based upon CORBA MASIF specification [4], e.g. Grasshopper [7, 13] are emerging. Agents using message-oriented communication either rely on an agent communication language like KQML [17] or FIPA [2] or use free-format communication. Several agent toolkits use KQML, e.g., JATLite from Stanford [35] and the AgentBuilder [1], a commercial product. Other commercial products, e.g., Aglets [28] or Voyager [21] use free-format communication. The Bond agent system uses KQML. Like other agent system we take advantage of few performatives supported by KQML, internally a KQML message is represented by a table consisting of attribute name and value pairs. This approach allowed us to add to our system XML-based communication with ease. At this time, Bond agents can mix KQML or XML messages.

Another important consideration in the design of an agent system is mobility. Agent systems like Aglets from IBM, [28], Telescript [40] from General Magic consider migration a defining property of an agent and a basic design goal is to allow an agent to migrate at any time. In Bond, migration is considered a rare event in the life of an agent and migration is possible only under certain circumstances and we implement a *weak migration model.* We restrict the locations an agent may migrate to and the time when migration is possible. A Bond agent runs under the control of the agent factory and uses the communication substrate provided by a Resident, thus it cannot migrate freely to any site [10]. An agent may move at the time of a transition from one state to another. If multiple planes are running concurrently then we have to wait until each plane reaches a transition.

To migrate an agent we simply send the blueprint of the agent and the model including the current state of the agent to a new site, [10]. The agent factory at the receiving end re-assembles the agent and when so instructed activates it from the state found in the model.

The statechart model [23] designed by David Harel is used to specify embedded systems and UML [32]-based systems. Our multi-plane agent state machine can be seen as a different way of expressing the parallelism which in statecharts are expressed as *concurrent sub-states.* On the other hand we have chosen to use a simpler state machine than those used in state charts. For example, in statecharts transitions can have conditions and actions associated with them, while in our model, only states can generate actions and transitions are unconditional. The theoretical foundation behind this decision is that in our model the structural component (the multiplane state machine), the active components (the strategies) and the model of the world are clearly separated. If we introduced a condition on a transition, that would clearly be a boolean function on the model, thus making the state machine dependent on the model. If an action were associated with the transition that would either imply that actions can be generated outside strategies, or alternatively that there is a strategy which is not determined by the state vector. Both of these semantics are expressed in our model by *inserting an intermediate node* between the source and destination, the strategy of these node than performing the desired action or evaluating the condition. Thus, the multi-plane state machines in our model can be larger for the same task than the corresponding statechart, but they are easier to analyze and generate, because of the simpler semantics. On the other hand, real time systems are easier to specify in the statechart format. Another feature of statecharts, the possibility to define embedded sub-states is also missing in our system. Its functionality in most cases can be replaced by the state vector of the multi-plane state machine. We are currently investigating the possible benefits of introducing sub-states in our model; although they improve the expressiveness of the system, they also introduce difficulties in implementing checkpointing, migration and agent surgery [12].

2.2 A Component-Based Architecture for Agents

Now we present a component-based architecture for software agents. In this architecture an agent consists of a group of active objects linked together by a data structure, rather than a large monolithic code. The behavior of the agent is determined by the active and passive objects and the data structure. The active objects usually consist of compiled code, thus can be executed with little additional overhead. The data structure can be modified with ease allowing for flexible behavior. The structure of the agent is presented in Figure 1. The four major components of an agent are: the model, the agenda, the state machines, and strategies.

The *model of the world* is a container object which contains the information the agent has about its environment. There is no restriction of the format of this information: it can be a knowledge base or ontology composed of logical facts and predicates, a pre-trained neural network, a collection of meta-objects or different forms of handles of external objects (file handles, sockets, etc), or typically, a heterogeneous collection of all these. The model also stores information the agent has about itself, e.g., plans or intentions if the agent conforms to the BDI model.

The *agenda object* defines the goal of the agent. The agenda implements a boolean function and a distance function on the model. The boolean function shows if the agent accomplished its goal or not. The agenda acts as a termination condition for the agents, except for agents with a *continuous agenda* where their goal is to maintain the agenda as being satisfied. The distance function may be used by the strategies to choose their actions.

The *multi-plane state machine* of the agent is a data structure composed of a number of state machines arranged in *planes*. The state of each state machine is defined by the active node. The state of the agent is defined by a *vector of states*. An agent changes its state by performing *transitions*. In turn transitions are triggered by internal or external *events*. External events are messages sent by other agents or programs. The set of external messages that trigger transitions of the agent's state machine defines the *control subprotocol* of the agent.

Each node of the multi-plane state machine has associated a *strategy object*. Strategy objects generate *actions* based upon the model and the agenda of the agent. Strategies do not reveal internal state information - their behavior is determined exclusively by the model and the agenda. The strategies must store their state in the model. Actions are considered atomic from the agent's point of view, external and/or internal events interrupt the agent only between actions. Each action is defined exclusively by the agenda of the agent and the current model. A strategy can terminate by triggering a transition or by generating an internal event. After the completion of the transition the agent moves into a new state where a different strategy defines its behavior.

The implementation we propose for the agent execution model is based on *strategies*.

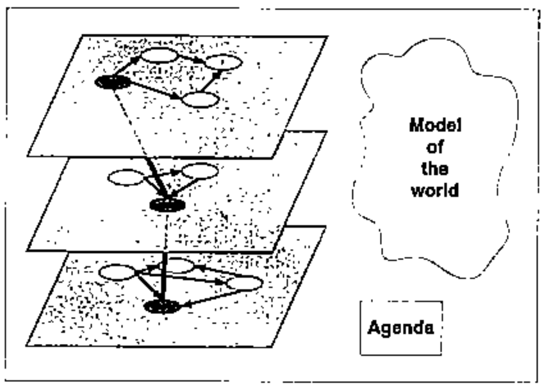

Figure 1: The multi-plane structure of agents

Informally, a strategy is a function which takes as parameters the model of the world and the agenda of the agent and returns actions. From the implementation point of view, a strategy is a Java object with a function called `action()` that performs the actions needed at the given instance.

The strategies are activated: (a) in response to external events and (b) as the flow of control requires while pursuing the agent's agenda. *Messages* from remote applications, and *user interface events* like pressed keys, mouse-clicks are examples of external events. The strategies are activated by the event handling mechanism - the Java event system for GUI events, or the messaging thread for messages in case of external events, or by an *action scheduler*.

The state of the agent is defined by a *vector of states*, which implies that the behavior or the agent is determined by a *vector of strategies.*

This structure allows us to assign different strategies for handling different types of events - for example a strategy from one plane handles the messages, while the other plane is handling the user interface events. One of the planes may provide reasoning or planning functions, one the execution, another one carry out housekeeping operations. The strategies in these planes are activated by the action scheduler.

The multi-plane structure provides the means to express concurrent agent activities. The actual nature of the parallelism is determined by the scheduling mechanism used by the action scheduler. In case of a round-robin activation mechanism the actions belonging to different strategies are interleaved without overlapping while multi-threaded execution allows for truly concurrent actions. Other possible activation schemes are priority-based and preemptive.

The agents are described by their *active components* (the strategies) and the *structural components* - the multi-plane state machine.

A strategy should be compatible with the agent implementation language, Java in case of Bond. There are two requirements a software component should meet to be a valid strategy: it should allow its state to be linked to the model and it should break its behavior into actions. JavaBeans, ActiveX objects, C++ libraries or functions in interpreted languages can all be valid strategies. In the Bond system, besides Java-written strategies, we are currently supporting strategies written in Jess and Python through the JPython interpreter. Any other language can be used through the Java native interface.

The structural component of an agent, the multi-plane state machine, can be constructed as a program, but a more flexible approach is a textual description, interpreted by an *agent factory*. An agent description language called *Blueprint* was defined to describe the structure of an agent. We are now extending the blueprint agent description language to accommodate

XML-based agent description. Other agent systems, e.g. Zeus [31], use a visual programming interface to create agents.

3 Work in Progress

3.1 Knowledge Management

Bond adopts the knowledge model provided by the Open Knowledge Base Connectivity Protocol [16]. This is achieved through integration with Protégé 2000, a knowledge modeling tool and programming library [22]. Each instance of the base class of the Bond object hierarchy can be viewed as an instance frame conforming to one or more class frames; subsequently the own slots associated with each Bond object represent assertions about the object or about the abstract entity represented by it. The uses of knowledge management abstractions in Bond include descriptions of capabilities of strategies and agents, representation of properties registered with community services, planning operator pre- and postconditions and data structures used for communication with the inference engine. Thanks to the facilities of Protégé, knowledge represented in a running Bond system can be exported in the Resource Description Framework format and new Bond objects can be assembled from imported RDF descriptions.

3.2 Agent Management. The Microserver.

Management, monitoring and debugging in multi-agent systems pose serious challenges. Agent debugging and monitoring should be done with as little intrusion as possible, agent management should be highly efficient, we need a uniform interface to access agent properties.

We propose to enable dynamic access to running agents and their properties using a microserver. A *microserver* is a light thread managing predefined *access-points* to enable external access through an appropriate protocol, e.g., HTTP, to agent's properties.

The *property access-point* enables public access to the agent properties. For example in case of distributed monitoring we can access the public name space from within the running Java Virtual Machine, or name spaces composed of variable properties which may be *.set()* and *.get()*.

The *method invocation access-point* enables RMI-like calls to any Bond object including agents. The task of the microserver is to translate from the internal object messaging protocol (inside the Java Virtual Machine) to the external (ex. HTTP) access protocol. The design of the formats and serialization of corresponding call arguments and results are especially challenging.

We implemented access-points corresponding to the Bond Object and the Bond Directory. Our implementation enables access to all objects registered with the local directory, access to dynamic and static properties of an object, and arbitrary calls to any method. The access points allow us to access the agent factory and to create and control Bond agents, to access the model and strategies of a running agent. Thus we are making significant steps towards distributed Web-based debugging of agents.

Bond supports several different communication mechanisms, such as reliable message delivery, best-effort message delivery and multicast message delivery. Each Bond object is able to send and receive messages using methods provided by the `bondCommunicator` class.

Upon initialization, an instance of *bondCommunicator* is created and available for object communication purposes. It is running as a separate *Bond messaging thread.* This thread implements all the function of message sending, receiving and delivering. Bond uses an abstraction called a *communication engine* to describe the interface between `bondCommunicator` and the communicating objects.

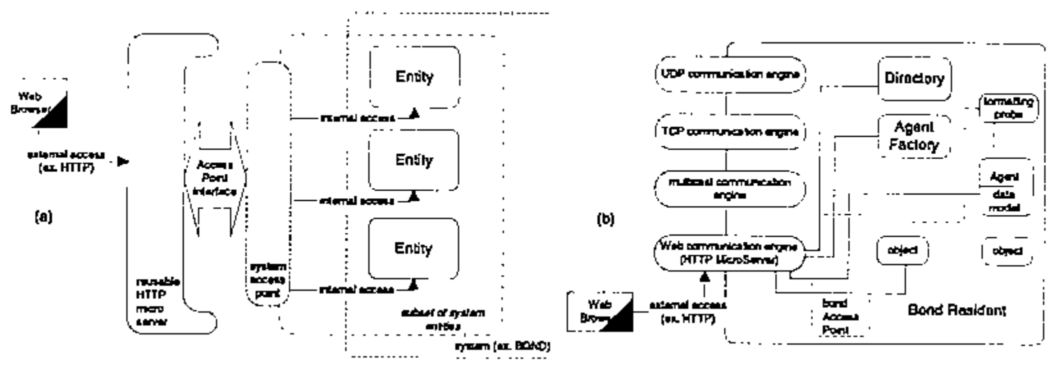

Figure 2: (a) The Microserver uses the access point interface implemented by the system access point in order to export properties of internal system entities. (b) The Microserver is integrated naturally as a communication engine in Bond. The Access points are corresponding to the agent factory, directory, agents and any other bond Object within the current Resident.

We integrated the HTTP microserver implementation as another communication engine in Bond. This approach is consistent with the overall Bond architecture. It allows direct browser access to any Bond object including the agent factory, as well as the ability to integrate the Bond framework with other systems using a microserver and a general purpose access-point.

Another interesting issues to explore is the deployment of probes to access a Bond object. Probes are objects attached dynamically to Bond objects to augment their ability to understand new sub-protocols [9], [11], and support new functionality.

In this case, either a direct microserver-based probe can be used, enabling the object to directly export it's own functionality or specialized probes may be designed in order to just enable the system's web communication engine (microserver) to access the object. This is subject to further design changes.

We implemented several general-purpose access-points, reusable to a great extent by any Java-based agent or distributed object system, e.g., the generic Java object field and the Java object method invocation access-points. These access-points are written as namespaces, implementing the NameSpace and AccessPoint interfaces.

The Java Object Reflection Field Accesspoint exports and allows access to any Java object's properties in a transparent manner (from the Object's perspective). It can be used to remotely monitor changes in the Object as seen in it's properties (variables, methods).

The Java Method Invocation Accesspoint exports callability of corresponding object's methods. This implies the ability to serialize call arguments as well as getting back method return. Issues such a protocol timeouts, stream serialization issues etc. arise but are partially solved in the current implementation.

Both access-points use the Java reflection mechanisms to discover object properties, fields and methods without involving the Object's code itself in this discovery. This is one of the strong points to be made about this approach. The initial designer of any system that is to be MicroServer enabled is not involved with issues pertaining to external access. In this respect the access-points can be reused as "observers" within any (currently Java based) system.

4 Applications

To test the limitations and the flexibility of our system, we developed several applications of the Bond agents ranging from a resource discovery agent to a network of PDE solver agents. We overview some of these applications.

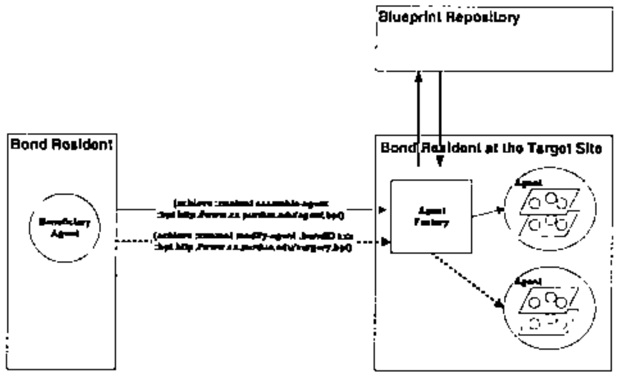

Figure 3: The dynamic deployment and modification of monitoring agents. The Beneficiary agent sends either a blueprint (solid line) or a surgery script (dotted line) to an agent factory to deploy a monitoring agent or to modify an existing one. The agent factory assembles it with strategies which are available from either local or remote blueprint repository

4.1 Resource Discovery

The Bond agents for resource discovery and monitoring have distinct advantages over statically configured monitors which have to be re-designed and programmed if they are deployed to other heterogeneous nodes. Moreover the local monitors should be pre-installed [18] [14] [6]. The dynamic composability and surgery of the Bond agents makes it possible to deploy monitoring agents on the fly with strategies compatible with target nodes, and modify them on demand either to perform other tasks or to operate on other heterogeneous resources.

We developed an agent-based resource discovery and monitoring system shown in Figure 3. Agents running at individual nodes learn about the existence of other agents by using *distributed awareness*, a distributed mechanism by which each node maintains locations of other nodes it has communicated with over a period of time and exchanges periodically this information among themselves [25]. Whenever an agent, a *beneficiary agent*, needs detailed information about individual components of other nodes, it uses the distributed awareness information to identify a target node, then creates a blueprint of a monitoring agent capable of probing and reporting the required information on the target node, and sends the blueprint to an agent factory of it. The agent factory assembles the monitoring agent with strategies comaptible on its node and launches it to work. A blueprint repository, which is either local or remote, stores a set of strategies. By sending a surgery script, the beneficiary agent can modify the agents as desired.

This solution is scalable and suitable for heterogeneous environments where the architecture and the hardware resources of individual nodes differ, the services provided by the system are diverse, the bandwidth and the latency of the communication links cover a broad range. On the other hand, the amount of resources used by agents might be larger than those required by other monitoring systems.

4.2 An Adaptive Video Server

Adaptive MPEG agent system implements an architecture supporting server reconfiguration and resource reservations for a video application [26]. Software agents provide feedback regarding desired and attained quality of service at the client side. Server agents respond by reconfiguring video streaming and reserving communication bandwidth and/or CPU cycles according to a set of rules. An inference engine, a component of the server agent, controls an

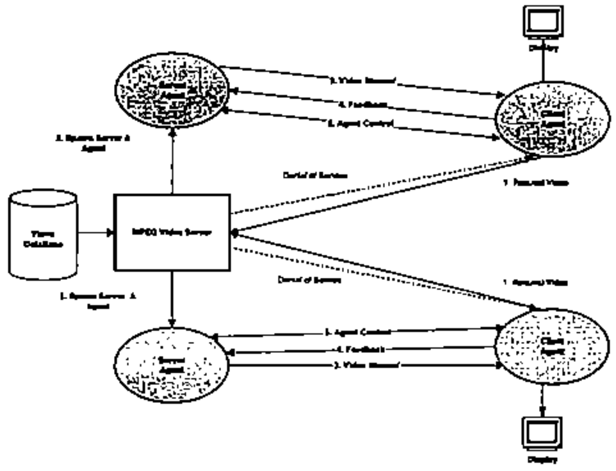

Figure 4: The MPEG system consists of a server and a set of server agents and client agents. The server and client agents support video streaming and display functions respectively. With a set of rules, the server agents can respond to changing resource state by adapting the video streaming.

adaptation mechanism. A native bandwidth scheduler and a CPU scheduler in Solaris 2.5.1 support QoS reservation

The architecture of the adaptive MPEG agent system is shown in Figure 4. An MPEG client agent is responsible for displaying a video stream and monitoring the reception of the video stream. When a client agent requests a video, an MPEG video server spawns an MPEG server agent which delivers and controls video streaming. Two communication channels exist between a client and its corresponding server side: a control channel for streaming commands and feedback from client to server, and a data channel for the streaming.

The MPEG server agent is configured to deliver an MPEG compressed video stream at start-up. As resource state of network bandwidth and CPU loads on the server and client change, the server agent can gracefully adapt by selecting one of three supported streaming modes: B/P frame dropping mode, Server decoding mode, and server decoding and dropping mode.

The application-specific program that adapts the current streaming mode to system resource state is written as a set of rules for the Java Expert System Shell (JESS) [20]. Our adaptation design is based on the following considerations. There are three resources on an end-to-end streaming path: server CPU, network, and client CPU, any one of which is a potential bottleneck limiting performance. Once a bottleneck is identified, one of the following adaptation rules reacts accordingly: bandwidth reservation rule, CPU reservation rule, dropping rule, decoding rule, and decoded dropping rule.

The advantage of the agent-based MPEG system is greater flexibility and system reconfigurability. The rules governing the behavior of the agents can be modified dynamically. Moreover, the agents themselves can be modified to add another streaming modes by the surgery.

4.3 Agent-Based Workflow Management

Motivated by deficiencies of existing workflow management systems (WFMS) in the area of flexibility and adaptability to change we initiated work on building a workflow management framework on top of the Bond system [33]. Usually in WFMS implementations agents enhance the functionality of existing WFMS and act as personal assistants performing actions on behalf of the workflow participants and/or facilitating interaction with other participants or the workflow enactment engine. We propose an agent–based WFMS architecture in which software agents perform the core task of workflow enactment. In particular we concentrate on the use of agents as *case managers*: autonomous entities overlooking the processing of single units of work. Our assumption is that an agent–based implementation is more capable of dealing with dynamic workflows and with complex processing requirements where many parameters influence the routing decisions and require some inferential capabilities. We also believe that the software engineering of workflow management systems is critical and instead of creating monolithic systems we should assemble them out of components and attempt to reuse the components.

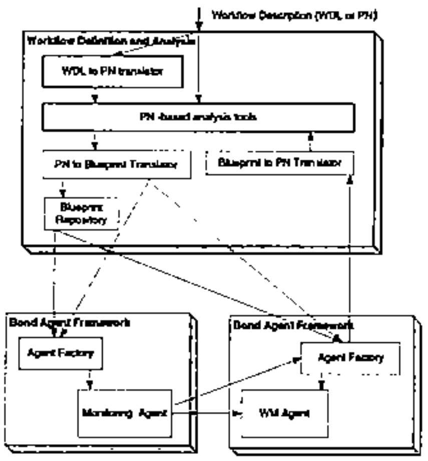

Figure 5: Workflow management in Bond

Figure 5 illustrates the definition and execution of a workflow in Bond. The workflow management agent originally created from a static description can be modified based upon the information provided by the monitoring agent. Several workflows may be created as a result of mutations suffered by the original workflow [38]. Once the new blueprint is created dynamically, it goes through the analysis procedure and only then it can be stored in the blueprint repository. The distinction between the monitoring agent and the workflow management agent is blurred, if necessary they can be merged together into a single agent.

We use Petri nets as an unambiguous language for specifying the workflow definition and provide a mechanism for enacting a large class of Petri net–based workflow definitions on the Bond finite state machine. For interoperatbility reasons we also supply a translator from the industry standard Workflow Process Definition Language [15] to our internal representation.

4.4 Remote Web Server Monitoring

The widespread use of web servers for business-oriented activities requires service guarantees because large variations of response time or even a very short time service failure can make an enormous negative economic impact as estimated in [24]. In addition to the service guarantee, the need for support for multiple classes of services based on QoS, various web-server farming, and security against denial-of-service will affect the future technologies related to web server configuration and management.

An agent-controlled management of web servers is under development. It can facilitate the tasks of web administrators. One typical example of those tasks is the following; based on the report from commercial web monitoring service companies [3] [5], the web administrators reboot failed servers, improve revealed bottlenecks, and optimize systems. Intelligent agents can automate these tasks with a set of rules describing conditions and corresponding actions, which mainly control parameters of servers.

An architecture of agent-controlled web cluster is shown in Figure 6. Actual web servers place behind a proxy, a frond-end server which dispatch requests to the web servers according to a *dispatching table* defined by a set of policies or load balancing algorithms. Only the proxy is exposed to outer world users. This is similar to other web cluster architectures except having an *adaptation agent* and *monitoring agents.*

The adaptation agent controls the web configuration by using intelligence simulated by a set of management rules provided by administrators, e.g. the dispatching table modification for load balancing, caching, QoS connection, and request filtering. The adaptation agent also communicates with its peers controlling other sites mirroring the service for global load balance and traffic optimization. The prominent advantage of the management by the adaptation agent is the flexibility coming from the separation of the agent from functional servers, while other web clusters use the static and monolithic management scheme.

The monitoring agents are mobile agents deployed to strategic places on the Internet to collect performance data of the web service by accessing the web service periodically. They can measure response time, test new transaction, monitor error or failure in the perspective of human users. The data provided by these agent are used by the adaptation agent to improve the web configuration. The monitoring agent is useful in the sense that they faciliate the external monitoring by which the bottleneck detection is possible because the server response time consists of communication time and processing time.

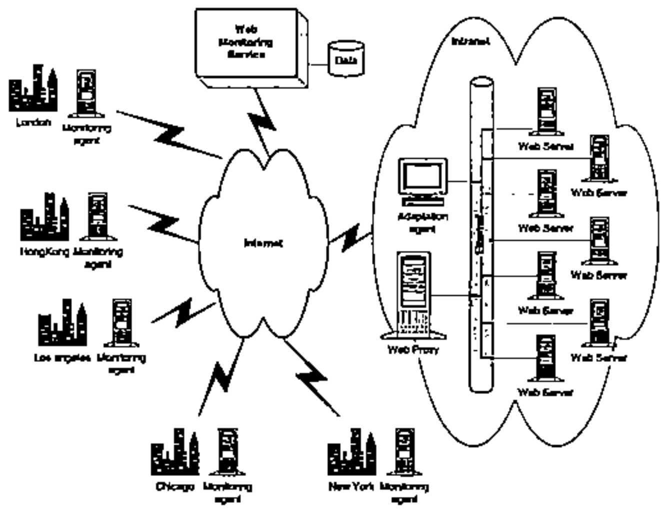

Figure 6: Agent-based web management and monitoring system. The adaptation agent is responsible for controlling a set of web servers. It manages a dispatching table for load balancing and QoS-aware web service. The monitoring agents collect web-performance data in the perspective of human users

At the same time, we propose to develop the following agents: an agent simulating the web-surfing behavior of human users for customer-oriented data, an *interface agent* for administrators to manually configure web cluster, and an agent controlling monitoring agents by directing the geographic distribution of monitoring agents, the temporal activation patterns of the monitoring agents, and the workload caused by the monitoring agents.

4.5 A Network of PDE Solver Agents

Data parallelism is a common approach to reduce the computing time and to improve the quality of the solution for data-intensive applications. Often the algorithm for processing each data segment is rather complex and the effort to partition the data, to determine the optimal number of data segments, to combine the partial results, to adapt to a specific computing environment and to user requirements must be delegated to another program. Mixing control and management functions with the computational algorithm leads in such cases to brittle and complex software. We developed a network of PDE solver agents and discussed its application for modeling propagation and scattering of acoustic waves in the ocean [37].

Agents with inference abilities coordinate the execution and mediate the conflicts while solving PDEs. Three types of agents are involved: one `PDECoordinator` agent, several `PDESolver` and `PDEMediator` agents. The `PDECoordinator` is responsible with the control of the entire application, a `PDEMediator` arbitrates between the two solvers sharing a boundary between two domains, and a `PDESolver` is a wrapper for the legacy application. Thus we were able to identify with relative ease the functions expected from each agent and write new strategies in Java. The actual design and implementation of the network of PDE solving agents took less than one month. Thus, the main advantage of the solution we propose is a drastic reduction of the development time from several months to a few weeks.

5 Conclusions

The agent system discussed in this paper can be positioned among the mainstream, multipurpose agent systems with several distinctive features:

- An agent implements a multi-plane state machine model.
- We introduce an agent description language called *blueprint* to specify the structure an agent. An agent factory translates a blueprint into an internal data structure controlling the run-time behavior of the agent and allows Bond agents to be modified at run time and to migrate.
- We pay close attention to the software engineering of agents and define a component-based architecture. Strategies and planes are reusable components. Our goal is to build an agent using off-the-shelf components.
- We aim to control and debug agents using a Web-based interface.

The flexibility of the design is illustrated by several applications discussed in Section 4.

6 Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

References

[1] Agentbuilder framework. URL `http://www.agentbuilder.com`.

[2] Fipa specifications. URL `http://www.fipa.org`.

[3] Keynote. URL `http://www.keynote.com`.

[4] MASIF - The CORBA Mobile Agent Specification. URL `http://www.omg.org/cgi-bin/doc?orbos/98-03-09`.

[5] Service Metrics. URL `http://www.servicemetrics.com`.

[6] Tivoli Enterprise Solutions. URL `http://www.tivoli.com/products/solutions`.

[7] C. Bäumer, M. Breugst, S. Choy, and T. Magedanz. Grasshopper — A universal agent platform based on OMG MASIF and FIPA standards. In Ahmed Karmouch and Roger Impley, editors, *First International Workshop on Mobile Agents for Telecommunication Applications (MATA'99)*, pages 1–18, Ottawa, Canada, October 1999. World Scientific Publishing Ltd.

[8] S. Bhattacharjee, K. L. Calvert, and E. Zegura. On Active Networking and Congestion. Technical Report GIT-CC-96-02, Georgia Institute of Technology. College of Computing.

[9] L. Bölöni and D. C. Marinescu. An Object-Oriented Framework for Building Collaborative Network Agents. In A. Kandel, K. Hoffmann, D. Mlynek, and N.H. Teodorescu, editors, *Intelligent Systems and Interfaces.* Kluwer Publising House, 1999.

[10] L. Bölöni and Dan C. Marinescu. A component agent model - from theory to implementation. In *Proceedings of the AT2AI Workshop, Vienna, Austria, April 2000, to appear.*

[11] Ladislau Bölöni, Ruibing Hao, Kyungkoo Jun, and Dan C. Marinescu. An object-oriented approach for semantic understanding of messages in a distributed object system. In *Proceedings of the International Conference on Software Engineering Applied to Networking and Parallel/ Distributed Computing, Rheims, France*, May 2000.

[12] Ladislau Bölöni and Dan C. Marinescu. Agent surgery: The case for mutable agents. In *Proceedings of the Third Workshop on Bio-Inspired Solutions to Parallel Processing Problems (BioSP3), Cancun, Mexico*, May 2000.

[13] M. Breugst, I. Busse, S. Covaci, and T. Magedanz. Grasshopper – A Mobile Agent Platform for IN Based Service Environments. In *Proceedings of IEEE IN Workshop 1998*, pages 279–290, Bordeaux, France, May 1998.

[14] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in legion. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing in conjunction with the International Parallel and Distributed Processing Symposium*, April 1999.

[15] Workflow Management Coalition. Interface 1: Process definition interchange process model, 11 1998. WfMC TC-1016-P v7.04.

[16] R. Fikes and A. Farquhar. Distributed repositories of highly expressive reusable knowledge. Technical Report 97-02, Knowledge Systems Lab Stanford, 1997.

[17] Tim Finin et al. Specification of the KQML Agent-Communication Language – plus example agent policies and architectures, 1993.

[18] S. Fitzgerald, I. Foster, C. Kesselman, G. Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–375, 1997.

[19] S. Franklin and A. Graesser. Is it an agent, or just a program? In *Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages*. Springer Verlag, 1996.

[20] E. Friedman-Hill. Jess, the java expert system shell. Technical Report SAND98-8206, Sandia National Laboratories, 1999.

[21] G. Glass. ObjectSpace voyager — the agent ORB for Java. *Lecture Notes in Computer Science*, 1368:38–??, 1998.

[22] W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, and M. A. Musen. Knowledge modeling at the millennium (the design and evolution of protege-2000). Technical Report SMI-1999-0801, Stanford Medical Informatics, 1999.

[23] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, 1987.

[24] Zona Research Inc. White Paper: The Economic Impacts of Unacceptable Web Site Download Speeds, 1999.

[25] K. Jun, L. Bölöni, K. Palacz, and D. C. Marinescu. Agent–Based Resource Discovery. In *Proceedings of Heterogeneous Computing Workshop 2000, to appear*, 2000.

[26] K. Jun, L. Bölöni, D. Yau, and D. C. Marinescu. Intelligent QoS Support for an Adaptive Video Service. In *Proceedings of IRMA 2000, to appear*, 2000.

[27] David Kotz and Robert S. Gray. Mobile code: The future of the Internet. In *Proceedings of the Workshop "Mobile Agents in the Context of Competition and Cooperation (MAC3)" at Autonomous Agents '99*, pages 6–12, May 1999.

[28] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, 1998.

[29] Dejan Milojicic. Mobile Agent Applications. *IEEE Concurrency*, 7(3), 1999.

[30] Hyacinth Nwana and Divine Ndumu. A perspective on software agents research. *The Knowledge Engineering Review*, January 1999.

[31] Hyacinth Nwana, Divine Ndumu, Lyndon Lee, and Jaron Collis. Zeus: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13 (1):129–186, 1999.

[32] Object Management Group. *OMG Unified Modeling Language Specification*.

[33] Krzysztof Palacz and Dan C. Marinescu. An agent-based workflow management system. In *Proc. AAAI Spring Symposium Workshop "Bringing Knowledge to Business Processes"*, 2000.

[34] M. Paolucci, D. Kalp, A. Pannu, O. Shehory, and K. Sycara. *Lecture Notes in Artificial Intelligence, Intelligent Agents*, chapter A Planning Component for RETSINA Agents.

[35] Charles Petrie. Agent-based engineering, the web, and intelligence. *IEEE Expert*, 11(6):24–29, December 1996.

[36] J. Smith, K. Calvert, S. Murphy, H. Orman, and L. Peterson. Activating Networks: A Progress Report. April, 1999.

[37] P. Tsompanopoulou, L. Bölöni, D. C. Marinescu, and J. R. Rice. The Design of Software Agents for a Network of PDE Solvers. In *Workshop on Agent Technologies for High Performance Computing, Agents 99*, pages 57–68. IEEE Press, 1999.

[38] W. M. P. van der Aalst and T. Basten. Inheritance of Workflows. An approach to tackling problems related to change. (draft).

[39] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE INFOCOM, San Francisco*, 1998.

[40] James E. White. Telescript technology: Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.