International Journal of Cooperative Information Systems

Supporting Distributed Autonomous Information Services Using Coordination

Avigdor Gal
Department of MSIS, Rutgers University, 94 Rockafellar Road
Piscataway, NJ 08854-8054, USA
and

John Mylopoulos
Department of Computer Science, University of Toronto, 10 King's College Road
Toronto, ONT M5S 3H5, Canada

Received (to be inserted
Revised by Publisher)

The large quantity and often questionable quality of available information in the information age provides a shaky foundation for decision making by individuals and organizations alike. This has created a tremendous demand for information services which can access, filter, process and present information on an as-needed basis. However, two factors complicate the design of such information services, namely the distributed and the autonomous nature of data sources.
This paper reports on the design and implementation of a generic architecture for supporting information services, which meets the above challenge. The architecture adopts concepts from conceptual modeling to offer a transparent description of the information sources' setting and uses active databases techniques to offer a declarative, event-based language for defining coordination rules for integrating distributed information services. Accordingly, the proposed architecture supports two of the most prominent utilities of information services, namely the pre-designed flow of operations and the reactive provision of information.
In addition to describing the architecture and illustrating its features with an example, the paper presents a prototype implementation and reports on some experimental performance results.

Keywords: Cooperative systems, Information Mediation, Advanced Data Models, Coordination Technologies

1. Introduction and motivation

Information plays a critical role in the information age by contributing to the fulfillment of goals for individuals and organizations alike. In order to ensure effective decision making, the information should be of high quality and modest quantity, achieved through suitable preprocessing and filtering of potentially vast amount of data. These demands have created the need for information services, capable of providing coherent and compact information obtained from distributed, autonomous data sources on an as-needed basis.

The distributed and autonomous nature of information services poses severe technical challenges for any technology that attempts to provide coherent and compact information. In particular, information needs to be gathered and filtered from a large, open and changing network of data sources. Moreover, these sources were developed independently, may be based on heterogeneous data models, and may contain duplicate or even contradictory data. The technical challenge of offering information services in such a setting is highlighted by the advent of the World Wide Web, its current status and its evolving nature.

This complexity of information services creates a need for an overall information services architecture to support their distributed and autonomous characteristics. An information services architecture should satisfy the following three desirable requirements in providing a suitable infrastructure for distributed autonomous information services:

Transparency: The architecture should provide a transparent description of the application domain. In particular, the architecture should allow a designer to abstract services and communication from geographically dispersed and heterogeneous components and to use them as building blocks of the application.

Flexibility: The architecture should be flexible enough to accommodate rapid modifications to information services. The need for rapid modifications evolves from the frequent changes to the data sources (e.g. modifications of Web sites).

Efficiency: The architecture should facilitate an efficient exchange of data and requests for services even under a heavy load of communication and a considerably large set of information services.

In this paper we report on the design and implementation of an architecture to support information services, which is particularly well suited for information services in autonomous and distributed computerized environments, and which meets the above mentioned requirements. We base our research on two observations that differentiate information services from query-based systems that support distributed databases. First, information services follow pre-designed (although flexible and continuously changing) methods for collecting, manipulating and releasing information. Second, in order to provide coherent and current information, information

services should take initiatives in supporting the active analysis of data. Towards this end, our research employs a novel approach that consists of two major components:

Coordinator: a set of tools and mechanisms that enable active monitoring, coordination and cooperation among data sources by using a rich semantic data model. Thus, the architecture is suitable for executing information services in a distributed, autonomous environment while supporting an active self-analysis of the continuously changing data sources.

Information repository: the part of the environment that manages information about the information services. Its main function is to facilitate data integration among the various components of the architecture and to provide a flexible tool for abstracting and modifying methods of operation. In addition, it implements the notion of "the current state" of the environment and stores historical data.

The main contribution of this paper is in proposing an information service architecture that fulfills the three requirements given above. This architecture is suitable for supporting the unique characteristics of information services, i.e. pre-defined methods and active behaviour. It uses an event-driven language with well-defined semantics that makes it easy to use and maintain rules, even by non-programmers. To support our claim for efficiency we present a performance evaluation based on a prototype implementation. These experiments show that the information services architecture performs reasonably well under various loads of the system, and that the coordinator does not add a significant overhead to the application's performance. As discussed in Section 2, the unique property of the presented model is twofold; it uses active rules to achieve the required behaviour of a distributed information service, rather than using passive, polling-based methods. Also, it enables the abstraction of information services by using a model that is rich in semantics, thus allowing a model transparency that separates the abstract description of the information service from its physical construction.

As a concrete motivating example we introduce the following case study which enhances the functionality of a legacy system by using a wrapping method[9]. A digital library retains electronic copies of books (each licensed and watermarked separately) and distributes them via an electronic network. Adopting a modus operandi of non-digital libraries, a registered copy of a book can be borrowed by one borrower at any given time point.[a]Therefore, a borrower is required to send the file back (or, alternatively, her privileges are being revoked) within a given time frame. The current implementation of the library does not provide user-friendly features. It uses a text-based mailer for communication with borrowers and it is neither capable of supporting a books waiting list, nor does it provide any assistance in registering new borrowers. These difficulties can be overcome by providing a wrapper that

[a]A similar approach was adopted by commercial digital libraries, e.g. netLibrary.[33]

suppresses certain existing behaviours of the application (e.g. a "try again later" response to a request for a book on-loan), enables others (e.g. sending a book to a borrower) and generates new behaviours (e.g. an "on waiting list" notification). The case study provides a typical example of a set of information services, some of which are considered to be black boxes[38] and therefore should be wrapped, while others can be modified and adapted to the new required functionality. As we show in this paper, the coordinator provides the architecture for building wrappers by using a rather simple language and a powerful abstraction mechanism supported by the architecture's repository.

The rest of the paper is organized as follows. In Section 2 we present related work. Section 3 introduces the architecture and presents its components in detail, including the information model, the communication model and the global coordination mechanism. A discussion of the architecture is provided in Section 4. Section 5 describes a prototype implementation of the architecture within the CoopWARE (Cooperation With Active Relationship Enforcement) project[31], and presents preliminary experimental results of the performance of the architecture under different workloads. Section 6 summarizes the contributions of this paper and points to directions for future research.

2. Background and related work

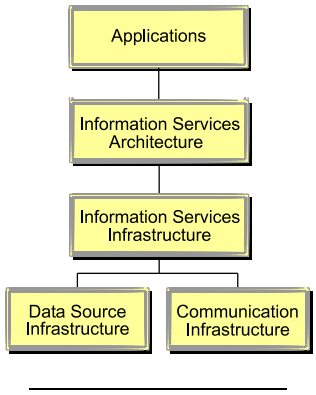

Fig. 1. The layered approach towards distributed heterogeneous information services

This section is aimed at positioning the proposed architecture in the broader context of cooperative information systems. As a conceptual framework we adopt a layered approach for the design and maintenance of information services.[b] Figure 1 illustrates such an approach where each layer is founded on the infrastructure provided by the lower layers. A short description of each of the layers follows:

[b] The layered approach was suggested as part of a tutorial on distributed heterogeneous information services, presented at CAiSE*98[18] by the authors.

- The data source infrastructure consists of various methods for managing the data quality of a single data source by using either syntactic (e.g. data mining) or semantic (e.g. metadata) tools. At this layer, techniques for extracting information from a wide variety of data sources, including databases, semistructured information systems[2] and flat files[1] are utilized.

- The communication infrastructure consists of the seven layers of the OSI reference model[35]. The higher information service layers build on the existing telecommunication technology.

- An information service infrastructure assumes the existence of communication and data source tools and provides mechanisms for handling information from heterogeneous distributed data sources. Existing tools and standards developed at this layer include ORB[20] in the CORBA model, HTTP requests, Java virtual machines embedded in Web browsers to run applets, and remote procedure call protocols.

- An information service architecture provides an overall architecture for handling the semantic issues involved in it. At this layer, tools for semantic understanding of heterogeneous and distributed data sources are developed in order to provide a transparent representation of the domain, which is independent of the underlying data sources. Several such architectures were developed by the research community, including TSIMMIS[12] and several works in the workflow area, e.g.[7]. A comparison of the proposed architecture with existing architectures is provided later in this section, yet as a motivating example for the need of an information service architecture consider a new interface for the digital library with two components, as follows. On the client side, a program is downloaded and provides a form-based tool for registering to the library and for borrowing books. The processing of the information is handled at the server's end and is transmitted to the coordinator for further manipulation. At the information service infrastructure layer, the interface is defined as two separate services that are distributed over the network. However, the information service architecture abstracts the description to adopt it to the organizational/user viewpoint where the interface is a single component.

- The top layer, the application layer, consists of particular classes of services, such as data warehouses, intranet applications, electronic commerce, and others.

The layered structure is useful in delineating research issues and in offering a useful method for integrating technologies. For example, one of the main difficulties in setting up coordinators in a distributed environment involves the heterogeneity of the interoperable systems in terms of issues such as hardware, syntax and semantics. The layered structure allows one to discuss solutions to the coordination problem independent of issues as hardware heterogeneity and syntactic interoperability by

using existing tools from the information service infrastructure, e.g. CORBA or ODMG. We allow for semantic interoperability capabilities at the application layer, by using semantic-rich models (e.g. Telos[30]) and semi-automatic concept generators (e.g. MIRROR[26]) provided by the information service architecture layer.

Integration is being examined in the areas of federated, distributed and heterogeneous databases. Interoperation and data integration techniques were suggested, e.g.[8,5,32] and[23]. Techniques for query decompositions were devised for structured[1,29] and unstructured data[37,36], and several query languages were suggested to support the distributed and heterogeneous characteristics of databases (e.g.[28]). Yet, these researches are query-based and therefore lack the active capabilities offered by the proposed architecture. Also, they provide little support for the design problem addressed in this paper, i.e. the continuous process of adopting information services to their environment.

Several proposals for integration architectures exist, including TSIMMIS[19] and SAP[14]. However, to the best of our knowledge, none of these architectures provide the flexibility offered by the proposed coordinator architecture, using the easy-to-use active language and the automatic interpretation of a high level language into an executable system (see Section 4 for details). The mediator concept (used in TSIMMIS) was introduced by Wiederhold[39] as a generic concept for providing information services. A *mediator* is a software module that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications. In this paper, we further refine the mediator concept, which consists of elements from the top two layers, by partitioning it into two components, where the information service architecture component is the main focus of this paper. Thus, we provide an architecture for an active instantiation of a mediator. It is worth noting that the proposed architecture fulfills the list of requirements as specified in[39] (i.e. small sized, simple, inspectable, accessible, flexible, and sharable), and therefore can serve as a provider of information services. The TSIMMIS project[19] uses mediators to manage multiple data sources, yet it suffers from the same drawbacks as do other models in the database research area, as it lacks an active support. Therefore, the TSIMMIS project can benefit from the proposed architecture by embedding it as a possible mediator's instantiation.

A different type of information services coordinator can be found in tightly coupled frameworks, such as SAP[14]. These frameworks are well suited for supporting coordination by using applications that were designed for cooperation, and therefore enable distributed computation in a homogenized environment. This type of framework, however, cannot support heterogeneous, autonomous, pre-existing data sources (legacy systems) and data services, thus restricting their applicability to more general settings.

Works in the research area of workflows provide tools for mapping business rules into coordinated process descriptions that can be executed automatically[13]. Therefore, workflows share many common aspects with the proposed coordinator. However, to the best of our knowledge, none of the existing workflow models pro-

vides an abstraction mechanism as powerful as the coordinator. For example, a unique property of our architecture is the representation of events as first class citizens, which allows the abstraction of the communication mechanism in addition to the abstraction of processes that most workflow models support. Thus, the coordinator's dynamic abstraction mechanism makes it possible to join processes, data items and events through classification, instantiation and generalization, while contemporary workflow models have far less expressive modeling capabilities.

A similar approach to the one taken in this paper is found in[34], where an agent technology is used and two types of agents are defined, namely tool agents and coordination agents. Nevertheless, the generalized approach taken in[34] (e.g. no specific details are given with respect to the structure of the coordination agent) prevents it from serving as a solution to the problem of continuously adapting the information services to the organisation's changing environment (which is supported herein by the coordinator's easy-to-use rule language). Also, the proposed architecture benefits the designer by enabling the use of active database analysis tools (e.g.[3,16]) and the automatic translation from a high-level definition into executable data structures (e.g.[15,16]).

The approach taken in[24] uses an event-driven approach to coordinate activities in an activity management system. Unlike the proposed model, their approach is cumbersome and the underlying language does not have clear semantics. Also, the ease of modifications cannot be evaluated based on the given description.

3. The integration architecture

In this section we present an information services architecture model that supports the characteristics as provided in Section 1. Figure 2 presents the general structure of the architecture. It consists of several components (a component can extend beyond the boundaries of a single machine) that communicate through a coordinator. Each component contains zero or more structured, unstructured or semistructured data sources, and has an *interface* library that defines a set of *services* that can be executed by the component associated with the interface and a set of *events*. A *coordinator* contains an *information schema*, a rule mechanism (reflected through the combination of "Rule Set" and "Rule Engine" in Figure 2), and an interface.

The following sections provide the details of the information services architecture: the information model (Section 3.1), the communication model (Section 3.2) and the global behaviour model (Section 3.3). A discussion of authorization issues in the proposed model is given in Section 3.4. The architecture is demonstrated using the definition of a wrapper for the digital library case study. The wrapper consists of two new components, as follows. A new interface programme enables the user to browse through the library's catalogue and to select a book to be sent to her, as well as to perform a registration process. This interface uses the library's catalogue, available as a world readable file, and assists users who are not familiar with the mailer in requesting and receiving books in a user-friendly fashion. Also, a

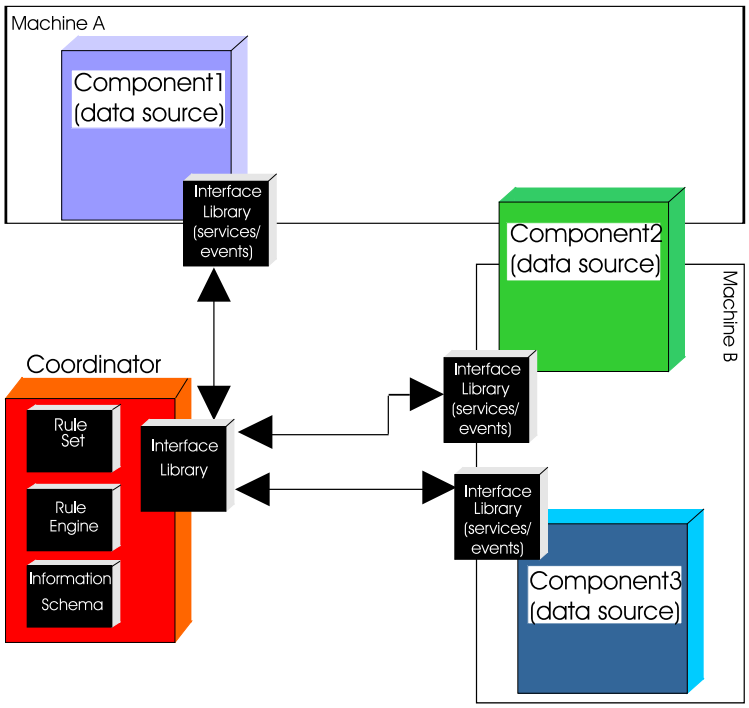

Fig. 2. The integration architecture

calendar application provides a book waiting lists. It enables the association of future activities with time and informs the coordinator once there is a need to perform a time-constrained activity.

3.1. The information model

The information model consists of three types of primitives. Firstly, *user* is a (virtual) person that takes part in designing the cooperation. The concept of a user is introduced as part of an authorization model for the application, as discussed in Section 3.4. Based on[6], a given user may belong to several groups (*roles*). The set of all roles is a partially ordered set R called a *role graph*. If a user belongs to a role, then she belongs to all the super-roles of this role, i.e. given two roles r_i and r_k such that $r_i \leq r_k, \forall u \in U(u \in r_i \Longrightarrow u \in r_k)$.

The second type of primitive is a *component*. Components are elements connected to the architecture. Each component is uniquely identified using the following five properties: accountID (account), machID (machine), procID (process), compClass (component class) and procStartTime, which is the machine local start time. At any given time, accountID, machID and procID uniquely identify a component. However, in order to maintain execution history and to enable the querying of the repository, the 5-tuple is required. A set of components can be defined using a partial description of the five characterizing properties. For example, a set of all the component instances that are on machine `Luna` and belong to a class `BookStack`. Each component belongs to a component class, which defines its possible behaviour (see services description below). All the instances of a component class share the same behaviour, although the underlying systems may differ. Therefore, the abstract behaviour of the `Mailer` is the same, whether it is Pine on a UNIX machine, Eudora on a Macintosh or any other mailer that is being used by the organization.

In a typical scenario, a single user provides all the information regarding a specific component, including the component's related services, events and messages. Once introduced, this user is considered the "owner" of the introduced component and may propagate her authorizations to other users subject to the integrity constraints of the specific authorization model.

Figure 3 presents the structure of the digital library information services architecture. The initial component set consists of two components, the `Mailer` and the `BookStack`. The wrapping process entails the use of two additional components, the `Interface` and the `Calendar`. The `Mailer`, `BookStack`, `Interface`, and `Calendar` are all components of the information services architecture. It is worth noting that the combination of the `Mailer` and the `BookStack` is a completely autonomous information service that should be treated as a black box (i.e. a single component rather than two separate components) and its internal communication and behaviour cannot be altered. Therefore, the coordinator can either enable or disable certain behaviours by controlling the input of the combined system. Also, since the `Mailer` is a public domain, its behaviour can be enhanced by using its common facilities.

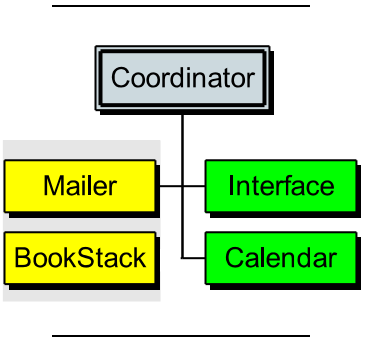

Fig. 3. The digital library information services architecture

The third type primitive of the proposed architecture is a *service*, an operation a component can perform. Services define the functionality of a component in terms of specific operations. Depending on the granularity of details given by the designer, a service can be refined into a network of internal subservices. A subservice is executed after the conclusion of its predecessors and before the execution of its successors. The use of subservices supports a range of methodologies, from black box to white box[38] and various shades of gray in between.

A service can have parameters that are instantiated at run-time. It also has a termination status, e.g. success, fail, and unknown. The unknown termination status is unique in the sense that it defines a scenario where a service termination status will not be available to the coordinator, and therefore cannot serve as a decision mechanism for further service activation. The service set can be partitioned according to the available knowledge with respect to a service termination status. A service is a TSA (Termination Status Available) service if its termination status is available and a Non-TSA service otherwise. The latter is accessed using monologue messages.

As an example, consider the services of the `Mailer` that are in use by the `BookStack`, as follows:

- `SendRequest:` ⟨`BorrowerID,BookID`⟩ sends a request for the book `BookID` as received from the borrower `BorrowerID`. The `Mailer` provides a termination status of success/fail for this service.

- `SendRequestAck:` ⟨`BorrowerID,BookID,Book`⟩ sends an acknowledgment for `BookID` to the borrower `BorrowerID` along with the electronic copy of the book. The `Mailer` does not provide a termination status for this service. It is worth noting that since the `BookStack` is the only component that holds the electronic books, this service cannot be requested by any other component.

- `SendTryAgain:` ⟨`BorrowerID,BookID`⟩ sends a message, informing the borrower `BorrowerID` that the book `BookID` is not available. The `Mailer` does

not provide a termination status for this service.

In addition, the wrapper extends the `Mailer` capabilities, in order to achieve the desired enhanced functionality. For example, we can add the following two services:

- `SendOverdueNotice:` ⟨`BorrowerID,BookID`⟩ is sent to the borrower `BorrowerID` as a reminder for an overdue book. The `Mailer` does not provide a termination status for this service.

- `SendOnWaitingList:` ⟨`BorrowerID,BookID`⟩ is sent to the borrower `BorrowerID` to notify her that she is on a waiting list for the book `BookID`. The `Mailer` does not provide a termination status for this service.

Since all communications to and from the `BookStack` are handled by the `Mailer`, the `BookStack` does not provide any services.

As mentioned above, the repository serves as a tool for maintaining the application's state. As such, the data that is stored in the repository typically consists of a set of status variables. For example, the legacy system that consists of the `Mailer` and the `BookStack` reacts to a request for a book that is on-loan by using the `SendTryAgain` service. In order to support a loan's waiting list, the coordinator should suppress this bahaviour and should generate an alternative behaviour that adds a name to a waiting list and sends a notification email to the user. Therefore, the repository maintains an availability list for each of the books in the library. The repository monitors any book request; if a book is available then the request is sent to the `BookStack`; otherwise, it is sent to the `Calendar` for further processing, using the service `AddToWaitingList:` ⟨`BorrowerID,BookID`⟩. Additional services of the `Calendar` are `SendToNextBorrower:` ⟨`BookID`⟩, where a returned book is sent to a borrower in the waiting list, and `RecordSignOut:` ⟨`BorrowerID,BookID`⟩, where a notification of a successful sign-out is sent for setting the book's return deadline. Generating new global behaviour is further discussed in Section 3.3.

3.2. The communication model

The communication model consists of two types of primitive elements. An *Event* is a compact reliable occurrence that enables the flow of information regarding the state of the information service. An event belongs to an event class, and is locally time stamped by the coordinator. Events are either external or internal, where external events are provided by external sensors, and internal events are provided by the components. Internal events can be related to the start or the end of a service/subservice. It is possible that the type of an event (external or internal, service/subservice, start/end) is unknown. An event instance that is related to a service or a subservice can be associated with a notification timing that specifies the relationships between the actual occurrence and the sending of the event instance. It can be immediate, deferred (with possible time boundaries), or unknown. An event can transfer parameters to be used by services.

An event is an abstraction of an activity execution in the application that is being brought to the attention of the coordinator, and serves as the main vehicle of changing the system's state reactively. Thus, an event can be an interrupt generated by the operating system, an email message that arrives at a local socket, or a trigger in a DBMS. Moreover, a single event can abstract several different activities, which allows the coordinator to treat them all as instantiations of the same class. Therefore, different activities may result in a similar response through the event abstraction.
For example, the `Calendar` provides the following two event classes:

- `BookOverdue:` ⟨`BorrowerID,BookID`⟩, a reminder that `BookID` borrowed by `BorrowerID` is overdue.

- `BookAvailable:` ⟨`BookID`⟩, a notification to the coordinator that the book `BookID` is available. This event is associated with the end of the `SendToNextBorrower` service by an immediate timing relationship.

The `Interface` provides the event `BookRequest:` ⟨`BorrowerID,BookID`⟩, a request for the book `BookID` by the borrower `BorrowerID`. Finally, the `Mailer` provides the event `BookReturned:` ⟨`BorrowerID,BookID`⟩, a notification of the return of the book `BookID` by the borrower `BorrowerID`. This event can be added to the current functionality of the `Mailer` by cc-ing the actual return of the book to the coordinator as well.

The second primitive element is a *message*, a communication element that enables the transfer of information among the components of the application and the coordinator. Each message has a Type, which can be one of the following: Request, Response, or Monologue. A request message is a message that is intended to be received, and responded to, by one or more components. A monologue message is similar to a request message except that there is no response expected by the sender of the message. A response message is a message that is intended to be received as a response to a request by a component. In addition to the message type, a message has a Source and a Destination, which can be a single component, a set of components, or a broadcast message. Messages also have a Priority, and of course Contents. It is worth noting that while messages serve as the underlying method for communication (including events and service requests), they also serve as a high-level element, useful for sending general types of information via the coordinator. For example, once a book is returned, the `Calendar` is required to send a request for the book on behalf of a borrower in a waiting list. Such a request can either be handled by the coordinator or be sent directly from the `Calendar`, via the coordinator, to the `Mailer`. Using the latter mode of design, the coordinator will pass this message without further interpretation. While this mode of design provides less information to the coordinator (as the message contents is not being analyzed by the coordinator), it increases the rapid transfer of information among the various components of the information service.

3.3. The global behaviour model

An application's global behaviour is defined using a set of rules. A rule is an event-driven atomic element that defines the cooperation among the application's components. Consistently with active database conventions, each rule consists of three elements, namely an event, a condition, and an action. An event component consists of a logic formula of events as defined above. A condition is a logic formula of data values, constants, etc. An action is either a set of services to be activated where each service contains the appropriate parameters, or an update to the application state as stored in the repository. The rule language is rich enough to define a rule concerning a variety of refinement levels, from a class of component instances through a subclass of component instances to a single component instance.

Although it is possible to have a rule where all the events and actions are part of the capabilities of a single component, we assume this case to be rare since the global behaviour is typically based on the coordination of several components. It is useful to identify three distinct types of rules (recognizing that hybrids also exist). *Component-specific rules* are rules whose action part involves a single component (while the event commonly belongs to another component), *cooperative rules* are rules whose action part involves several components, and *coordinator rules* are rules that define the coordinator's patterns of behaviour. While component-specific rules are typically given as part of the component class definition, cooperative rules are a result of design decisions of the application, and therefore are ordinarily user-defined.

Appendix 1 provides an example of a simple rule language that is utilized as part of the CoopWARE project. The language interpreter enforces a rapid response time to an event by enforcing the condition component to use only variables available within the repository. The language does not provide general active database mechanisms such as coupling modes (which are under construction), yet its simplicity does not interfere with the design of the applications we have experimented with, including the digital library example provided throughout the paper. The following set of rules serves as an example of a definition of the global behaviour of an application.

Event:	`BookRequest(BorrowerID,BookID)`
Condition:	`BookID.Status`$= 0$
Action:	`SendRequest(BorrowerID,BookID)`
	`RecordSignOut(BorrowerID,BookID)`

If a book is requested and is available, then the request is sent via the `Mailer` to the `BookStack`, and the `Calendar` is notified. This rule is an example of a cooperative rule.

Event:	`BookRequest(BorrowerID,BookID)`
Condition:	`BookID.Status`$\neq 0$
Action:	`AddToWaitingList(BorrowerID,BookID)`
	`SendOnWaitingList(BorrowerID,BookID)`

If a book is requested and is not available, then it is added to the waiting list (a `Calendar` service) and a notification is sent to the user (a `Mailer` service).

Event: `BookReturned(BorrowerID,BookID)`
Action: `SendToNextBorrower(BookID)`

If a book is returned, the `Calendar` is requested to generate a request for the first borrower on the waiting list. This rule is an example of a component-specific rule.

Event: `BookAvailable(BookID)`
Action: `BookID.Status`$:= 0$

An event from the `Calendar` to the coordinator changes the application's state by changing the book's `BookID` status from "not available" (`BookID.Status`$\neq 0$) to "available" (`BookID.Status`$= 0$). This rule is an example of a coordinator rule.

Event: `BookOverdue(BorrowerID,BookID)`
Action: `SendOverdueNotice(BorrowerID,BookID)`

An event from the `Calendar` to the coordinator triggers an overdue email (using a `Mailer`'s service) that is sent to the borrower `BorrowerID` regarding the book `BookID`.

It is worth noting that the process that results from a book return, i.e. choosing the next borrower on the waiting list and sending her the returned book, is done through the use of the message mechanism, and therefore is not captured by the rule set as given in this section.

3.4. Authorization

The definition of users and roles in the architecture provides the infrastructure for the authorization model[11]. A role is identified with each component, and the role graph corresponds to any hierarchy that can be defined on the set of components. Any instantiation of a component c identifies this instance to belong to the role associated with c. Therefore, any instance of c is authorized to perform any service defined for c and is authorized to signal any event defined for c.

In an organizational setting, the global behaviour model is a result of the cooperation of several users as reflected in the application's components cooperation. Therefore, a rule that invokes a service of a specific component needs to be authorized by the component's owner, her delegates, or any other user who is authorized based on the specific authorization model. For example, if access to a component c is authorized for a role r_k, then by definition any user in a role $r_i \leq r_k$ may define a rule that invokes a service of c. It is worth noting that cooperative rules involve the access of several components and therefore require the authorization of a group of users that, combined together, have the authorization to access all of the components in the rule. Therefore, the rule's authorization model enforces the required cooperation for generating a global behaviour.

The authorization for modifying the global behaviour is particularly important as the coordinator's rules can serve as a tool for malicious users to plant Trojan horses and to open covert channels. To prevent security bridges, we enforce the following authorization rules:

1. The coordinator is assigned with an independent role. Any user of this role

can manipulate the coordinator's global behaviour (although cooperative rules require the consent of all parties for a modification). Due to the sensitive status of the coordinator, any user assigned with this role should be a trusted user.

2. Component-specific rules of a component c can be provided by any user who is part of the role associated with c. Such a rule is of transient nature and thus, the services will be requested from the specific instantiation that provided the rule. Whenever a Multi-Level Security (MLS) authorization model is enforced, such rules are legitimate as long as the component providing the event is not at a higher security level than the component that provides the rule.

4. The architecture's properties

At this point, the process by which an individual information service introduces its capabilities (e.g. services and events) and its needs (i.e. rules) is known. It is also clear how a collection of information services can be pieced together in a consistent fashion to form a distributed autonomous information service. This process can now be evaluated against the characteristics, requirements and observations, as laid out in Section 1, to manifest its adequacy.

4.1. Transparency

The distribution of information is handled through communication in various levels, from messages, through the specification of events for supporting an active system, to the definition of rules for coordinating activities in a distributed system. A prominent communication feature of the proposed architecture is the use of events as a vehicle for managing the information service. While modern DBMSs support a triggering mechanism, many legacy data sources are not event-driven by their nature. However, the components need not be event-driven in order to be successfully combined into the architecture. Events serve as abstractions of any asynchronous output provided by a component for the purpose of triggering the coordinator rules. Therefore, if a component X provides a termination status of either "success" or "fail," the component's interface can translate the "success" message into an event of type Y, and the "fail" message into an event of type Z. Hence, both event-based systems and non-event based systems can be freely combined into the architecture for generating new information services.

The representation of events as first-class citizens is adopted from the area of active databases and allows the abstraction of the communications mechanism. For example, assume that the architecture handles two digital libraries, similar in functionality yet distinct in their platforms. Consequently, there may be two different Calendar instances in use, based on the programs available for each platform. While conceptually both Calendar instances provide the same event sets, the particular underlying messages may differ. Therefore, the proposed architecture adds the

abstraction of a communication mechanism for purposes of analysis and easy extension of the application, to the abstraction of services that is available in commercial WFMSs (WorkFlow Management Systems).

The transparency property is obtained via the abstract definition of components and services in addition to events. For example, consider a situation in which the BookStack switches from a Pine mailer to a Eudora mailer. While some modifications are required at the infrastructure layer, e.g. adding tools for reconstructing big files that were decomposed by the Eudora mailer, the abstract functionality of the application (including the definition of events) is left unchanged.

The handling of heterogeneity is facilitated by the adoption of the Telos information model[30], which is object-oriented in the sense that a Telos repository consists of objects ("propositions" in Telos terminology) and everything is an object, including classes, metaclasses and attributes. Moreover, the model strongly supports generalization, classification and attribution. The inheritance mechanism built within Telos is strict (i.e. a subclass cannot override the description of its superclasses) and allows for multiple inheritance. Likewise, multiple classification is allowed, in the sense that an object (a token, a class, or an attribute) can be an instance of several classes. Finally, attributes in Telos are multi-valued and can have their own attributes, since they are treated as first-class objects. These features make Telos particularly well suited for applications that involve evolving schemata and metamodeling[21]. In particular, these features make it possible to classify an information object under different views, corresponding to the systems that are being integrated through the proposed architecture.

Yet, a major drawback in using Telos to define a common schema for several integrated systems involves the complexity of building Telos meta-schemata and in populating them. To alleviate this problem, we have developed a methodology and a prototype system called MIRROR that facilitates the creation of the meta-schemata and their subsequent population. MIRROR is described in further detail in[26] and its integration with CoopWARE in[27].

4.2. Flexibility

An important novelty of our approach lies in the architecture's flexibility, which can be observed in several dimensions. First, the architecture provides a user-friendly modification mechanism of an information service, where modifications are propagated directly to coordinator and repository updates. To demonstrate this, consider the partial summary of the attributes that are defined for the various elements of the architecture in Table 1. These attributes are maintained in the repository, and are utilized at run-time to validate their applicability for activities in the coordinator's environment. Any change in an element results in a modification in the repository and in an immediate modification of the information service

[c] We use the term *information model* in the same way as *data model* is used in databases, i.e. a model that provides a set of data structures and associated operations as well as constraints for representing information.

Table 1. Associated attributes of the architecture elements (partial)

Element	Associated attributes	Element	Associated attributes
User	User-Name Role	Event	Event-Name Parameters External/Internal Related-Services Notification-Timing
Component	accountID machID procID compClass procStartTime	Message	Type Source Destination Priority Contents
Service	Service-Name Parameters SubServices Termination-Status Related-Events	Rule	Event Condition Action

architecture. Hence, the user-friendly interface of the information service architecture (see Section 5) and the high-level rule language of the coordinator jointly enable the designer to evaluate the consequences of any modification to the application. Moreover, there is no need for a re-compilation of the coordinator after such modifications.

Second, the use of an active technology, as opposed to the common passive, query-based approach, provides an excellent environment for defining the active behaviour of an information service. The use of rules further enables a designer to define in advance, as well as on-the-fly, methods for generating coherent and reliable information.

Third, the abstract description of the application enables model independence, where upgrades of existing systems do not necessarily result in modifications to the application's description. For example, the architecture provides an event-driven language as a semantic framework for collaboration and coordination among information services. This property of our architecture lays the grounds for a profound understanding of the architecture of information services and thus protects it from semantic failures due to frequent modifications and autonomous evolutions of its components. Therefore, the communications infrastructure can be CORBA-based, Internet-based, or any other communications infrastructure tool that allows the passage of messages among components. No matter which communications infrastructure model is used, or whether a change has occurred in the communications infrastructure, the abstract description of the application should not be modified as long as there are no changes to the enterprise model.

Finally, the architecture supports a flexible interaction mechanism that supports various levels of autonomy on the components' behalf. The architecture does not claim full ownership on the communication channels among components. Therefore, components can interact outside the coordinator environment without causing

any system instability. Also, components can use the communication infrastructure that the architecture provides to transfer messages. While the semantics of these messages is not available to the coordinator, the architecture provides a fast and robust environment for such activities. It is worth noting that whenever a component is allowed to utilize alternative channels of communication, some inconsistencies may occur and can be prevented by a careful design of the application. For example, assume that the Mailer and the BookStack of the digital library example may accept requests from borrowers, independent of the interface. In such a case, the coordinator does not have accurate information regarding the availability of books. Therefore, an alternative design of the system should be considered. Such an alternative can introduce the coordinator as a borrower of the library. In this case, a request for a book through the interface is translated into a book request by the coordinator, and the outcomes of such a request are propagated to the borrower based on the business rules provided towards the end of Section 1.

4.3. Efficiency

The use of an event-driven management for a distributed environment enhances the efficiency of the architecture, being more efficient than other methods (e.g. polling) in terms of both application resource utilization and responsiveness[25]. Also, it is conceived to be a more natural programming vehicle than a general-purpose third generation programming language. Furthermore, the language interpreter enforces a rapid response time to an event by enforcing the condition component to use only variables available in the repository, thus enhancing the architecture's efficiency even further. Section 5 provides several simulation runs to demonstrate the architecture's feasibility and efficiency.

5. Implementation and experiences

This section discusses implementation issues (Section 5.1) and provides simulation analysis and performance evaluation of the coordinator prototype (Section 5.2).

5.1. Implementation

A prototype of the coordinator, named CoopWARE, was implemented in C++, using the XLc compiler and a Web-based interface on IBM AIX Version 3 for RISC System/6000. It is supported by the Telos repository[30], implemented in C++ as well, and uses ODIs object-oriented database ObjectStore for persistence. The architecture prototype uses the communication model infrastructure as illustrated in Figure 4. It uses an extended version of TMB (Telos Message Bus). The TMB, implemented in C++, is an extensible message server through which all the components can communicate, both with the coordinator and with each other. These messages form the basis for all communication in the system. The message server has been implemented on top of *mbus*, an existing public domain software bus

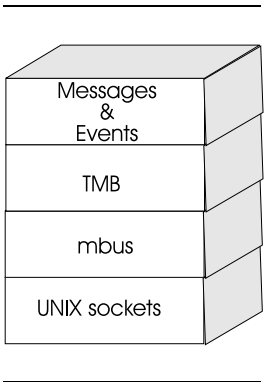

Fig. 4. The message layer structure

technology.[10] *Mbus*, in turn, is dependent on the presence of UNIX sockets. Since these protocols have been ported to virtually all modern computer platforms, including DOS/Windows and Apple machines, our architecture can incorporate components running in heterogeneous environments. An alternative communication infrastructure may use either CORBA or the Internet as more generally available tools. The use of the Internet as the communication infrastructure is currently under implementation.

Using TMB, messages can be transferred simultaneously to the coordinator and to its destination (in case the destination is not the coordinator). This mechanism enables efficient message passing in the case that there is no need for an additional interpretation on behalf of the coordinator. In addition, it prevents extensive modifications to the component instances themselves.

The registration interface is implemented using dynamic HTML. It allows a designer to define applications, components, services, messages, events, and rules. Through the interface, a designer can generate, update and delete any of the architecture's elements in an easy-to-use manner. For example, Figure 5 provides a screen snapshot of a service form. The form identifies all of the required information about a service, and it uses combo boxes for selecting from related lists of applications, components, and events. Figure 5 provides an example of registering the `SendToNextBorrower` service. The service is offered by the `Calendar`, which sends a termination status for it. The service receives one parameter (`BookID`) and it possibly generates the event `BookAvailable`. A submitted form is processed using the mSQL database and its associated Lite script language[22]. Once all of the required information is available, the new information is compiled into a Telos set of definitions, which replaces the existing one. It is worth noting that the term "compilation" refers to the Telos set of definitions, a process that can be done without recompiling the coordinator, and therefore any change will be swiftly embedded within the running system.

Fig. 5. A registration form

The interface is implemented as a skeleton library (skel.a), a set of routines which provides a common interface for communication among components and the coordinator. This library, in turn, uses a set of generic libraries that enable basic types of communication. The generic libraries are utilized by a designer to enable the flow of events, requests, and responses to and from the components. The Socket Interface (sockInt.a) library enables communication by connecting (client) or binding (server) to a socket. The Email Interface utilizes the Socket Interface to connect to the SMTP port (25) of a UNIX machine in order to send email messages and serves as an additional form of communication that can be utilized by a component.

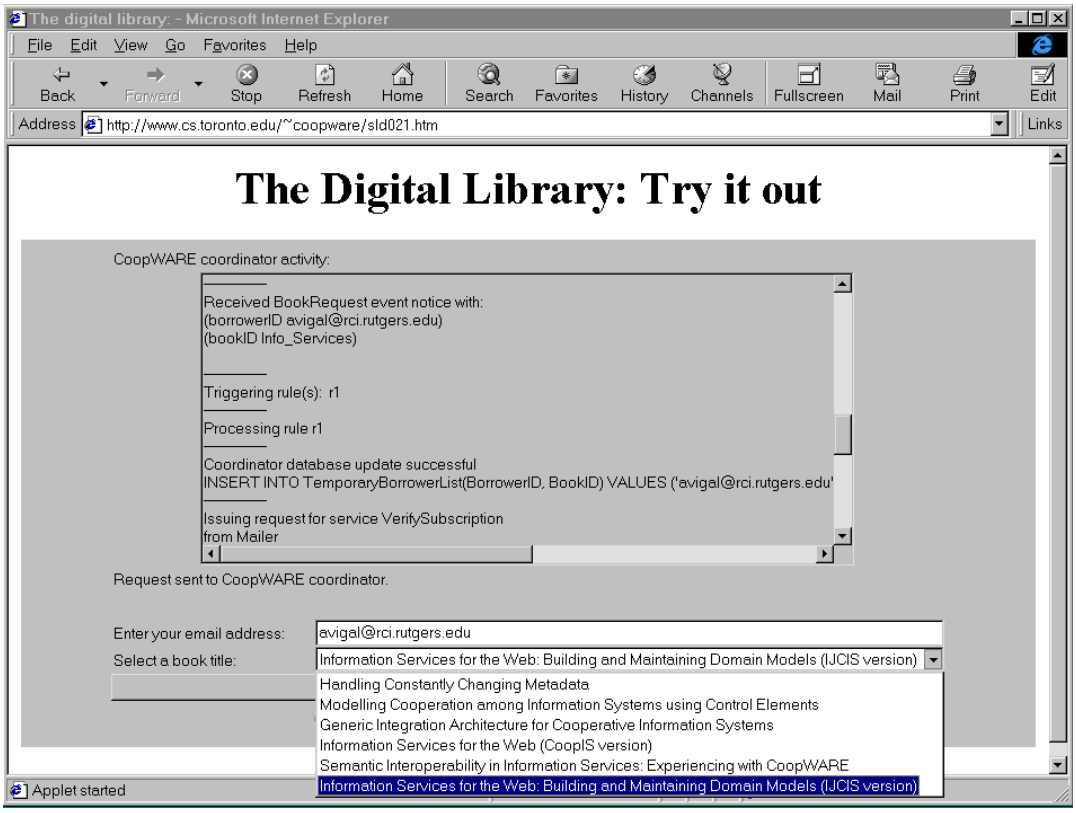

Fig. 6. The WWW Interface for the digital library

We have designed a Web-based prototype[17] to demonstrate the coordinator's capabilities, using the digital library case study. The prototype combines an Internet-based communication (between a Java applet version of the `Interface` and the coordinator) with a TMB communication to connect other parts of the application with the coordinator. Figure 6 presents the Interface through which a borrower

can connect to the digital library and can post requests for books (in this case, CoopWARE-related publications) from a given drop-down list.

5.2. Simulations

This section reports on a set of simulations aimed at evaluating the performance of the coordinator. In order to do so, we have isolated the impact of the coordinator's execution from other factors, such as the load of the network or the execution time of the components. Therefore, this section is primarily concerned with the coordinator performance, rather than general concerns in performing computation in a distributed environment. To isolate the impact of the coordinator's execution, we restricted the simulation to consist of monologues, rather than service requests. The latter entails a response that involves parameters external to the coordinator, such as the load on the network, and the performance of the components.

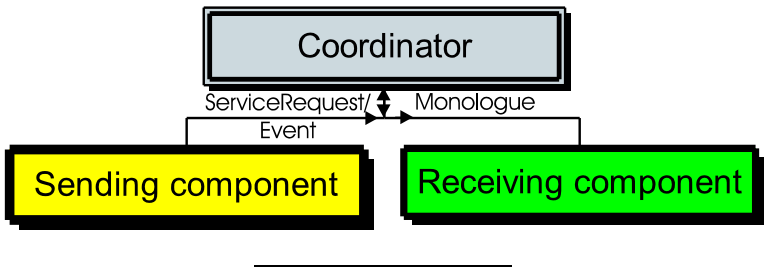

Fig. 7. The simulation model

Figure 7 describes a performance evaluation experiment, to evaluate the feasibility of the coordinator. A sending component sends periodically (every t_r seconds) either a request for a service or an event (depending on the simulation type), which is translated into monologue messages that are sent from the coordinator to the receiving component. A request for service requires the verification of the existence of the requesting component and the requested service. The coordinator should also locate the component that provides the service. Finally, the coordinator sends the message to that component. All activities beside the latter involve the use of the information given in the repository. An event requires the verification of the existence of the sending component, and of the existence of the event type in the repository. The coordinator retrieves all the rules that are triggered by the event, evaluates their conditions and activates services if a condition is evaluated to be true. As in a service request, all activities beside the latter involve the use of the information given in the repository. In the ensuing experiments we estimate the costs related with the coordinator's execution under various scenarios.

The simulation consists of several runs. In each run n_r service requests (or events) are sent and the average service time (t_s) is computed. $t_s = t_d - t_a$ is

computed as the time from the arrival of the request/event to the coordinator from the sending component (t_a) until all monologue messages that are related with the service/event are sent to the receiving component (t_d). This performance metric takes into account the coordinator's performance, regardless of the network or the components' performance. For each run, a distinct t_r is set where t_r can either be a constant or $t_r \sim \exp(t)$. The parameter t is set separately for each run of the simulation.

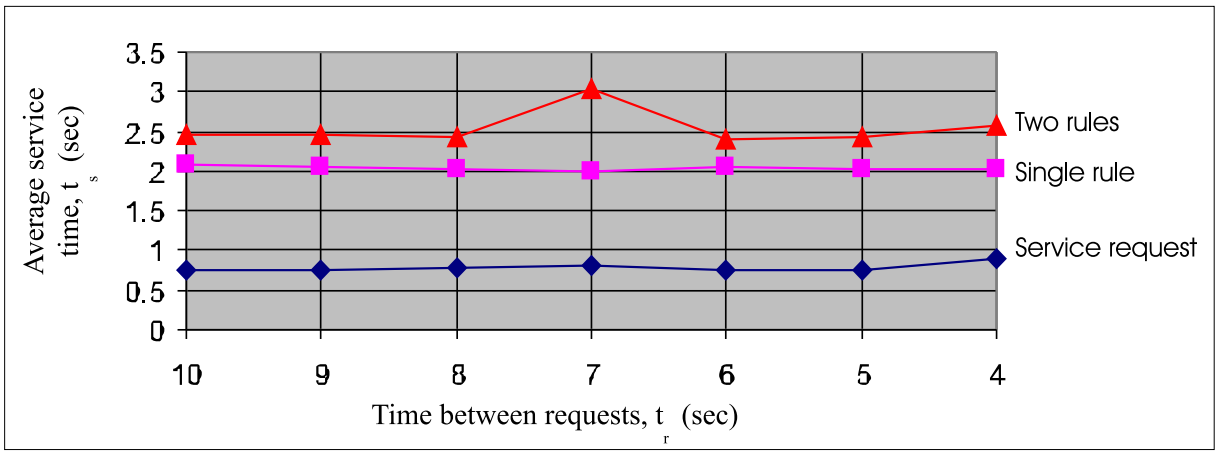

Fig. 8. Average service time (t_s) vs. t_r

Figure 8 presents the results of three simulation runs with $n_r = 1000$ and a constant time gap between consecutive messages. In one of the simulation runs (identified as *service request* in the graph), a service request was sent from the sending component to the receiving component via the coordinator, resulting in no rule activation. In the other two runs, an event was sent to the coordinator, resulting in monologue messages to the receiving component. The latter two runs differ in the number of processed rules. The *single rule* simulation describes a simulation with a single rule while the *two rules* simulation describes a simulation with two rules. In both cases, each rule results in a single monologue message. For each simulation run, a distinct $t_r \in \{4..10\}$ was selected and the average service time was recorded. The results show no significant variations in the average service time while using any t_r in the designated range. The average service time ranges from 0.74 seconds to 0.88 seconds for a service request, from 1.99 seconds to 2.07 seconds for an event with one processed rule, and from 2.41 seconds to 2.47 (with one exception) for an event with two processed rules. Thus, due to the rapid handling of each request/event and the gap between consecutive service requests, which is wide with respect to the coordinator's performance, there is actually no delay in processing a request.

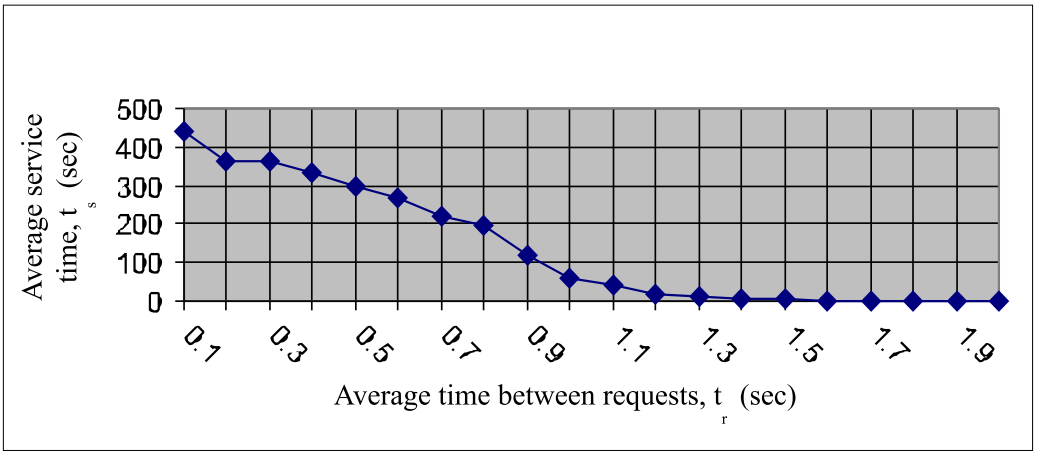

Fig. 9. Average service time (t_s) vs. $t_r \sim \exp(t)$

To analyze the behaviour of the coordinator under a heavy load of communication, we provide a simulation run with $n_r = 1000$ service requests, and $t_r \sim \exp(t)$ such that $0.1 \leq t \leq 2$. Figure 9 presents the result of this simulation run. The average service time ranges from 1.79 seconds (when $t = 2$) to an unreasonable time of 438.95 seconds (when $t = 0.1$). The shorter the average time between requests is, the longer the service time is, due to long waiting periods in the various queues. However, as long as the average time between requests is more than 1.3 seconds, the performance of the coordinator is reasonably good (up to 14 seconds per request). It is worth noting that while this experiment shows a reasonable performance for a prototype, careful optimization of the coordinator would enable better performance, even under an extreme load (less than 1.3 seconds).

An important performance issue deserving attention involves the coordinator's capability to handle simultaneous requests. To test this capability, the coordinator was bombarded with $n_r \in \{50..1000\}$ messages simultaneously. Figure 10 presents the recovery time, i.e. the length of time required for the coordinator to complete the handling of all n_r simultaneous requests. The main result of this simulation is that the coordinator maintains a constant processing time of any single request even under a heavy load, where for $n_r = 50$ the coordinator has recovered within 23.8 seconds and for $n_r = 1000$ it took the coordinator 376.87 seconds to recover. Therefore, if the coordinator faces a sudden increase in the rate of incoming requests, it is able to handle it in a reasonable time frame. Hence, the coordinator can easily recover from small amounts of simultaneous requests, and does not crash even under a very heavy load. An optimization of the coordinator, aimed at minimizing the recovery time by hard-wiring the connection between the coordinator and the repos-

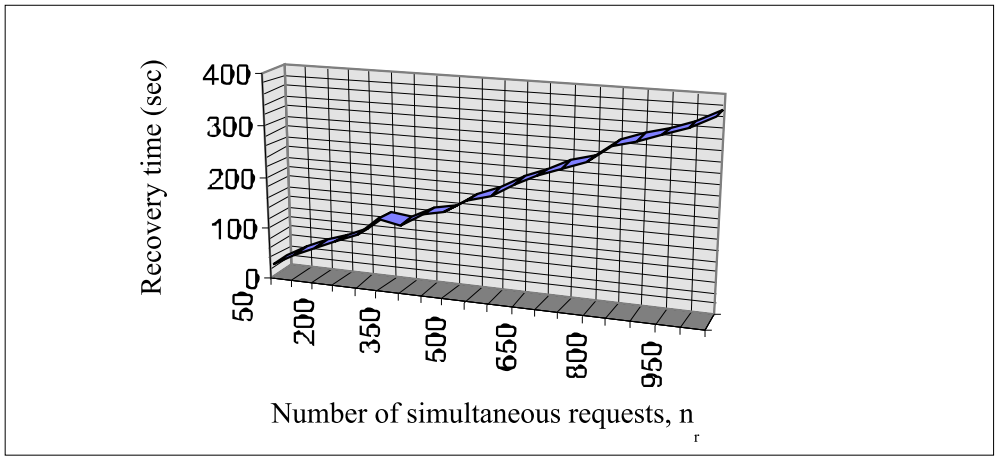

Fig. 10. Recovery time (t_s) vs. n_r simultaneous requests

itory rather than by using TMB, is currently under way. A possible architecture for handling pick times involves the use of backup coordinators that assist the original coordinator whenever the load of messages gets beyond an application-dependent threshold.

Figure 11 examines the effect of the number of processed rules (i.e. rules such that their condition is evaluated to be true) on the overall performance of the coordinator. Three different scenarios were examined, the first with 10 triggered rules, the second with 9 triggered rules and the third with 8 triggered rules. For each scenario, the number of unprocessed rules (n_{up}) was modified in the range of 0 to the maximal number of triggered rules, and the average service time (t_s) was measured. In all three scenarios the coordinator displayed a moderate increase in t_s with the increase in the number of processed rules (an average of approximately 1.5 seconds per each additional processed rule). Hence, the coordinator can handle a large number of concurrently triggered rules independent of the actual number of processed rules.

To conclude, we have found the coordinator's implementation to be robust under various scenarios, including heavy traffic and a moderate number of rules. Therefore, we believe this implementation provides an appropriate proof of concept. The coordinator, serving as a middleware, adds a reasonable overhead to the overall distributed processing.

6. Conclusion

We have presented the design and implementation of an architecture for the support of information services, which is based on the premise that information ser-

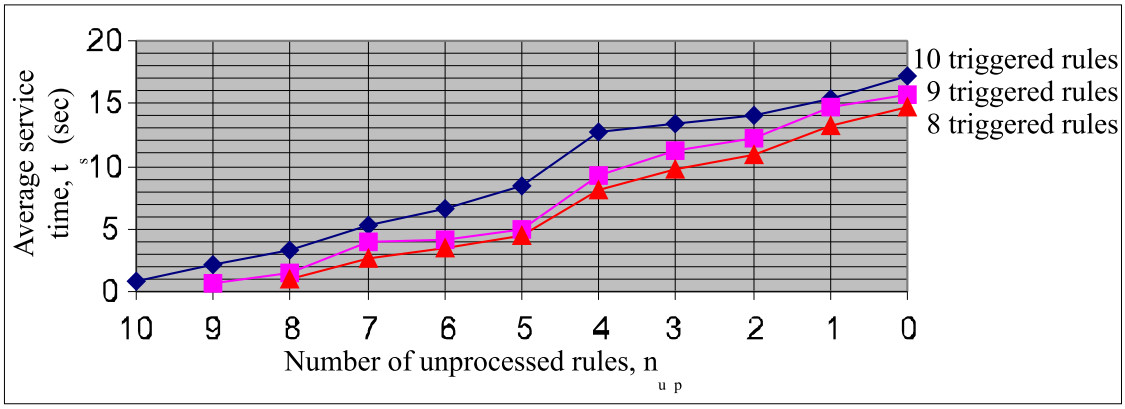

Fig. 11. Average service time (t_s) vs. n_{up} unprocessed rules

vices are mostly legacy components integrated through the architecture. In order to provide an optimal architecture for information services, we have adopted an active approach to processing information services, keeping in mind that information services follow pre-designed (although flexible and continuously changing) methods for collecting, manipulating and releasing information, and should take initiatives in order to provide coherent current information. The architecture supports facilities for the data integration of the components and for mechanisms for coordination among these components. It provides a high-level definition and manipulation language that is flexible enough to accommodate rapid modifications to services.

In addition to the proposed architecture, the paper reports on a prototype implementation of the architecture and several simulation runs with the prototype. The coordinator prototype supports an efficient exchange of requests for services and processing of rules even under a heavy transportation load of messages.

A major pitfall of integration architectures involves the initial cost of setting up an application. This cost consists of two types of processing, namely data integration and components' interfacing. *Data integration* involves homogenizing the terms that are used for transferring parameters and reaching a consensus glossary. It is a human-intensive process that involves domain experts as well as information systems experts.[d] *Components' interfacing* involves the modifications required to connect the interface with the application. The techniques for generating a component interface range from a wrapper to modifications to the original code, depending on the nature of the component that ranges from a black box to a white box. In Section 5 we have provided an overview of our own experience in building such interfaces. Both processes are hard to conduct, yet unavoidable in any system

[d] See[4] for methodologies of data integration in a database context.

that coordinates distributed systems. The advantage of the proposed architecture becomes apparent when considering the cost of change. Changes in the proposed model are straightforward due to the high level language of the coordinator that provides a transparent implementation of the domain, and therefore the cost of change is relatively low. The changes to the components are incremental and are built on the experience gained in the initial stage. In many cases, while changes are frequent, they are based on existing functionality and use the same environment. Therefore, such a change is translated into minor coordinator modifications due to the event-driven language, and into minor changes to the components that are relatively easy to implement after the initial setup. Therefore, the usefulness of the architecture should not be measured based solely on the initial cost. Rather, it should be evaluated over time, as in the long run the architecture provides a useful tool for aligning the organization's information system with the continuously changing enterprise model.

Several issues require further research. In[31] we have presented an initial description of an automatic translation of the coordinator's language into an executable system that is based on a control graph. A control graph is an executable data structure that coordinates the execution of various information services. The graph encapsulates the causality relationships among information services and provides analysis capabilities. Current research deals with implementing a full-scale system based on control graphs with enhanced analysis capabilities. Also, the design of a transaction mechanism for the rule execution engine and its implementation is currently under way. Finally, we would like to study the co-existence of several coordinators within one system, each of which manages a partial information repository and supports only some coordination and policy enforcement activities. This forms a distributed environment on an even higher level, and will possibly increase the performance of the overall system.

Acknowledgment

We would like to thank Martin Stanley and Kostas Kontogianis for their active assistance in the design of the model, and James Wysynski and Richard Murray who implemented the prototype of the coordinator. We would also like to thank Alex Borgida for suggesting the digital library domain as an example for the use of active rules and repositories in cooperative information systems and to Christina La for her contribution to the design of the registration forms. We are grateful to the anonymous reviewers for their useful remarks.

Appendix A The EBNF form of the event-driven language's grammar

The EBNF form of the event-driven language's grammar is as follows.

Rule	::=	*Event:* Event-Clause
		[*Condition:* Condition-Clause]
		Action: Action-Clause
Event-Clause	::=	Event-Clause ∨ Event-Clause
	|	Event [*(* Param {, Param}* *)]*
Condition-Clause	::=	[Sign] Cond-Unit {Connective [Sign] Cond-Unit}*
Action-Clause	::=	{Routine-Call-Exp}*
	|	Action-Unit
Event	::=	Identifier
Param	::=	Data-Element
Sign	::=	∼
Cond-Unit	::=	Data-Element Binary-Pred Constant
Connective	::=	∧
	|	∨
Routine-Call-Exp	::=	Routine-Name [*(* Param {, Param}* *)]*
Action-Unit	::=	Data-Element := Constant
Identifier	::=	LABEL
Data-Element	::=	Identifier
Binary-Pred	::=	=
	|	<
	|	>
	|	≤
	|	≥
	|	≠
Constant	::=	{x | x is a number}
	|	{"x" | x is a string}
Routine-Name	::=	Identifier

References

1. S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proceedings of the 19th International Conference on VLDB*, pages 73–84, Dublin, Ireland, 1993.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1), 1997.
3. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems, Lecture Notes on Computer Science, 985*, pages 165–181. Springer Verlag, 1995.
4. C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.
5. E. Bertino. Integration of heterogeneous database applications through an object-oriented interface. *Information systems*, 14(5):407–420, 1989.
6. E. Bertino and H. Weigand. An approach to authorization modeling in object-oriented database systems. *Data & Knowledge Engineering*, 12(1):1–29, February 1994.
7. Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *ACM SIGMOD Record*, 22(3), Sep 1993.
8. Y. Breitbart, P. Olson, and G. Thompson. Database integration in a distributed heterogeneous database system. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 301–310, Los Angeles, CA, Feb 1986.
9. M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the RIDE-DOM workshop*, pages 124–131, 1995.
10. A.M. Carroll. *ConversationBuilder: A Collaborative Erector Set.* PhD thesis, University of Illinois, 1993.
11. S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison-Welsley, 1994.
12. S. Chawathe, H. Garcia-Molina, H. Hammer, K. Ireland, Y. Papakonstantinon, J.D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of IPSJ*, Tokyo, Japan, 1994.
13. A. Chichoki, A.S. Hela, M. Rusinkiewicz, and D. Woelk. *Workflow and Process Automation: Concepts and Technology.* Kluwer Academic Publishers, Boston, MA., 1998.
14. T. Curran. *Client/Server Development With SAP's ABAP/4 Development Workbench 3.0.* Prentice Hall, July 1996. ISBN 0135253950.
15. O. Etzion. PARDES — a data-driven oriented active database model. *ACM SIGMOD Record*, 22(1):7–14, Mar 1993.
16. A. Gal. *TALE — A Temporal Active Language and Execution Model.* PhD thesis, Technion—Israel Institute of Technology, Technion City, Haifa, Israel, May 1995. Available through the author's WWW home page, http://www.cs.toronto.edu/∼avigal.
17. A. Gal and J. Mylopoulos. The CoopWARE demo: wrapping up a legacy system. http://www.cs.toronto.edu/∼coopware, 1997.
18. A. Gal and J. Mylopoulos. Distributed heterogeneous information services. Tutorial notes, CAiSE*98, 1998. available upon request from avigal@rci.rutgers.edu.
19. H. Garcia-Molina et al. The TSIMMIS approach to mediation: Data models and languages. In *Proceedings of the 2nd International Workshop on Next Generation Information Technologies and Systems (NGITS'95)*, pages 185–193, Naharia, Is-

rael, 1995.
20. The Object Management Group. The common object request broker: Architecture and specification. Technical Report 91.12.1 Rev 1.1, Object Management Group, December 1991.
21. M. Jarke, R. Gallersdoerfer, M. Jeusfeld, M. Staudt, and S. Eherer. A deductive object base for metadata management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995. Special Issue on Advances in Deductive Object-Oriented Databases.
22. Brian Jepson and David J. Hughes. *Official Guide to Mini Sql 2.0*. John Wiley & Sons, 1998.
23. K. Karlapalem, Q. Li, and C. Shum. HODFA: An architecture framework for homogenizing heterogeneous legacy databases. *ACM SIGMOD Record*, 24(1):15–20, Mar 1995.
24. K. Karlapalem, H.P. Yeung, and P.C.K. Hung. CapBasED-AMS—a framework for capability-based and event-driven activity management system. In *Proceedings of the Thirs International Conference on Cooperative Information Systems (CoopIS'95)*, pages 205–219, 1995.
25. M.J. Katchabaw, S.L. Howard, A.D. Marshall, and M.A. Bauer. Evaluating the cost of management: a distributed applications management testbed. In M. Bauer et al., editors, *Proceedings, CASCON*, pages 29–41, 1996.
26. S. Kerr. Data integration and change management using active metamodels. Master's thesis, University of Toronto, 1998.
27. S. Kerr, A. Gal, and J. Mylopoulos. Information services for the web: Building and maintaining domain models. In *Proceedings of the Third IFCIS Internationla Conference on Cooperative Information Systems (CoopIS'98)*, pages 4–13, NYC, NY, August 1998.
28. A. Levy, A. Rajaraman, and J. Ordile. Querying heterogeneous information sources using source description. In *Proceedings of the International Conference on VLDB*, pages 251–262, Bombay, India, 1996.
29. L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous information sources. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS'96)*, pages 140–156, Brussels, Belgium, June 1996.
30. J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4), October 1990.
31. J. Mylopoulos, A. Gal, K. Kontogiannis, and M. Stanley. A generic integration architecture for cooperative information systems. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, pages 208–217, Brussels, Belgium, June 1996.
32. S.B. Navathe et al. A federated architecture for heterogeneous information systems. In *Proc. 1989 Workshop on Heterogeneous Databases*, Northwestern University, Evanston, IL, December 1989.
33. netLibrary Web site. netlibrary.com.
34. M.C. Norrie and M. Wunderli. Tool agents in coordinated information systems. *Information Systems*, 22(2/3):59–77, April 1997.
35. International Standard Organization. The OSI reference model. http://www.iso.ch.
36. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructure heterogeneous information. In *International conference on Deductive and Object Oriented Databases*, 1995.
37. D. Suciu. Query decomposition and view maintenance for query languages for un-

structured data. In *Proceedings of the International Conference on VLDB*, pages 227–238, 1996.

38. G. Valetto and G.E. Kaiser. Enveloping sophisticated tools into computer-aided software engineering environments. In *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering (CASE'95)*, pages 40–48, 1995.
39. G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.