Dissemination Protocols for Event-Based Service-Oriented Architectures

Brahim Medjahed, *Member*, *IEEE*

Abstract—The event-driven, or notification-based, paradigm has attracted much research interest in areas such as distributed systems, databases, workflow, and grid computing. However, little attention was devoted to event-driven service-oriented architectures (SOAs). In this paper, we propose a novel framework for event-based interactions in SOAs. First, we introduce various notification patterns for event-driven SOAs. We define two taxonomies for dissemination protocols in SOAs: The interaction taxonomy identifies the different models through which Web services interact with each other and the filtering taxonomy classifies the events and services involved during dissemination. Second, we propose a dissemination pattern called implicit notification. In contrast to publish-subscribe, implicit notification does not require consumers to explicitly subscribe with producers. We define a model for implicit notifications and introduce a family of protocols for enabling this pattern. Finally, we describe a prototype implementation for a disaster management case study and conduct experiments to assess the performance of the proposed protocols.

Index Terms—Advanced services invocation framework, collaboration exchange protocol, collaborative services delivery platform, Web services interoperability.

1 Introduction

Service-oriented architecture (SOA) is a new approach for designing software systems by providing services to either user applications or other services distributed in a network via published and discoverable interfaces [1], [36], [37]. Although SOAs can be implemented using different technologies such as J2EE, CORBA, and JMS, the most common realization of SOAs is based on Web services [36]. Web services are loosely coupled entities that provide predefined capabilities via XML-based standards (e.g., WSDL, SOAP) on the Web [1]. They interact with each other through the exchange of messages. The most common interaction pattern used in Web services is request-response: A client submits a request to a server (e.g., request for quote from a stock Web service); the server sends back a reply to the client. However, Web services within an SOA often want to receive messages (also called *notifications*) when *events* occur in other services and applications [22]. Examples of events include computation results, applications data, status updates, errors, and exceptions.

The event-driven, or notification-based, interaction pattern is widely used to *disseminate* information among different entities in distributed systems [35]. It typically realizes the well-known publish-subscribe scheme [15]: Consumers register their interest in an event and are subsequently asynchronously notified of events generated by producers. Publish-subscribe is at the heart of message-oriented middleware and event-driven systems. Combining event-driven systems with SOAs, known as *event-driven SOAs*, inherits the features of both: Web services address the interoperability issue in heterogeneous distributed systems and event-driven systems enable asynchronous interactions. Events are disseminated in SOAs for various purposes, such as logging (e.g., for recovery, nonrepudiation), monitoring (e.g., for service availability), auditing (e.g., for privacy), and sending application-specific data (e.g., weather alerts, stock price changes). Event-driven SOAs are slated to play a key role in empowering vital areas such as disaster management, e-science, supply chain, and finance [13], [41].

The event-based paradigm has attracted much research interest in areas such as distributed systems [2], [8], [14], databases [11], [25], workflow [20], and grid computing [18]. However, little attention was devoted to event-driven SOAs. WS-Notification [21], [42] and WS-Eventing [9] are two competing standardization efforts for enabling event-based SOAs. Techniques such as [34] and [37] define notification protocols for Web services. These techniques and standardization efforts simply adapt the traditional publish-subscribe scheme to SOAs. In this paper, we propose a novel framework for event-based interactions in SOAs. We give below a summary of our contributions:

- We introduce various dissemination (or notification) patterns for event-driven SOAs. We show that these patterns complement each other; each pattern is suitable to particular scenarios (e.g., push versus pull, wireless versus wired). We define two taxonomies for dissemination protocols in event-driven SOAs: The *interaction taxonomy* identifies the various ways through which Web services interact with each other and the *filtering taxonomy* classifies the events and services (consumers and producers) involved during dissemination.

• The author is with the Department of Computer and Information Science, University of Michigan—Dearborn, 4901 Evergreen Road, Dearborn, Michigan, 48120. E-mail: brahim@umd.umich.edu.

Manuscript received 11 Nov. 2008; revised 24 Nov. 2008; accepted 26 Nov. 2008; published online 10 Dec. 2008.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2008-11-0101.
Digital Object Identifier no. 10.1109/TSC.2008.13.

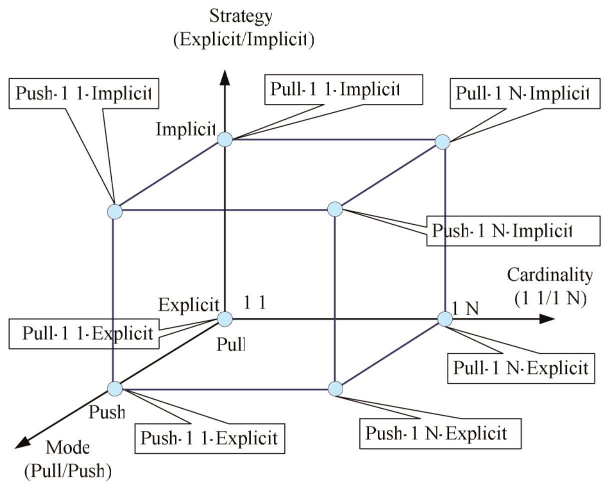


Fig. 1. Interaction taxonomy.

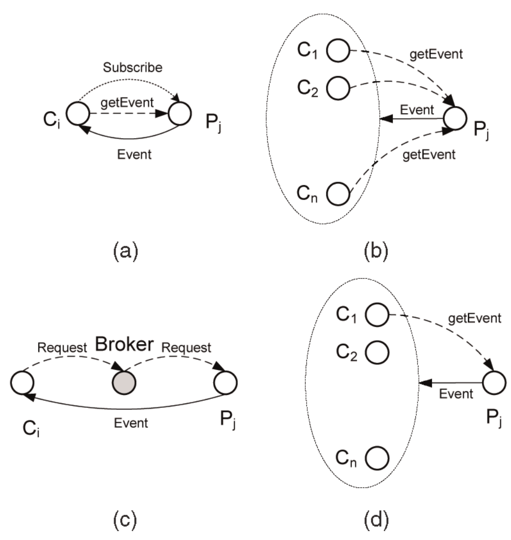


Fig. 2. Scenarios for pull-based dissemination. (a) Pull-1:1-Explicit. (b) Pull-1:N-Explicit. (c) Pull-1:1-Implicit. (d) Pull-1:N-Implicit.

- We describe a novel pattern called *implicit notification.* In contrast to publish-subscribe, implicit notification does not require consumers to explicitly subscribe with producers. At the reception of an event, a service determines the list of relevant *neighbors* (using a *distributed ontology*) and forwards the event to them. We propose a family of protocols for enabling the implicit notification pattern.
- We describe a prototype implementation and experiments to assess the viability and performance of implicit notification for a disaster management case study. Disaster management systems provide strategies to rapidly assess conditions, make decisions, deploy relief and resources, and respond effectively to crises (e.g., earthquake and hurricane) [4]. A recent report authored by the US National Research Council's Committee on "Using Information Technology to Enhance Disaster Management" identifies SOAs as one of the research directions for improving disaster management [33]. The adoption of event-based SOAs for disaster management helps deal with rapidly changing situations such as terrorist attacks and hurricanes [12].

The rest of this paper is organized as follows: In Section 2, we describe the interaction and filtering taxonomies for dissemination techniques in SOAs. In Section 3, we present the implicit notification model. In Section 4, we propose protocols for enabling the implicit notification model. In Section 5, we describe a prototype implementation of the implicit notification model for a disaster management case study. Section 6 is devoted to performance study. In Section 7, we overview related work. In Section 8, we present concluding remarks.

2 Classification of Dissemination Techniques in SOAs

In this section, we introduce two taxonomies for categorizing dissemination protocols in event-based SOAs: the *interaction* and *filtering* taxonomies.

2.1 Interaction Taxonomy

The *interaction taxonomy* (Fig. 1) identifies the different models through which services interact with each other in event-driven SOAs. It categorizes those models in a 3D space: *mode*, *cardinality*, and *strategy*.

The *mode* (pull versus push) dimension specifies whether dissemination is initiated by consumers or producers. In the pull strategy, events are delivered by producers as a reply to specific requests (e.g., getEvent) submitted by one or more consumers. In the push strategy, events are sent by producers without prior requests from consumers. *Cardinality* (one-to-one versus one-to-many) defines the number of consumers to which information is delivered. One-to-one (1:1) cardinality means that a producer is delivering information to one single consumer. One-to-many (1:N) cardinality expresses that a producer is transmitting information to several consumers. It is important to note that one-to-many is *not* many one-to-one [15]. We identify two techniques for enabling 1:N cardinality: broadcast and multicast. In broadcast, producers send information over a medium on which consumers can listen. One example, in wireless environments, is that of a mobile service (producer) broadcasting the requested information via a wireless channel [43]. In multicast, information is sent to a specific set of consumers. Some of the first systems to enable multicast communication were based on the Isis framework [7]. The *strategy* (implicit versus explicit) specifies whether the consumer made an explicit request for information dissemination (e.g., by sending a request or subscribing to events) or not (i.e., implicit). The strategy and mode dimensions are orthogonal. As illustrated in the rest of this section, both push-based and pull-based disseminations may be explicit and implicit.

Pull-based dissemination—As depicted in Fig. 2, there are four pull-based dissemination patterns for Web services. In the *Pull-1:1-Explicit* pattern, C_i requests a specific event (through getEvent) from P_j; P_j replies by sending the event to C_i (Fig. 2a). We identify two scenarios corresponding to this pattern. First, C_i has a subscription with P_j on the event and has the possibility of polling P_j (through getEvent) at regular intervals and pulling events, if any are available. The second scenario is a variant of the observer design pattern defined in [19]. C_i has a subscription with P_j on an event; once the event is available, P_j sends minimum event information (e.g., availability of a multimedia file) to C_i; C_i then query for the rest of the information (e.g., get the file from P_j).

In the *Pull-1:N-Explicit* pattern (Fig. 2b), several (subscribed) consumers explicitly request information from a producer. The producer replies to all consumers using multicast communication. The Atomic Commitment Protocol [1], [36] is a scenario where this pattern could be used; different services (that belong to the same transaction) send requests to a coordinator which will reply by atomically sending "abort" or "commit" to all services via multicast communication.

The *Pull-1:1-Implicit* pattern (Fig. 2c) refers to the scenario where a consumer C_i sends a request R to a broker. The broker selects the producer P_j to execute R. The selection is based on predefined parameters such as quality of service (QoS) and consumer's profile. Then, the broker forwards R to P_j. Later on, P_j sends events related to R to C_i. For example, C_i may send a request for airline reservation to a travel agency (i.e., the broker). The broker decides to submit a reservation request to a given airline's Web service P_j. At anytime, P_j may send events (e.g., booking confirmation and flight cancellation) to C_i.

In the *Pull-1:N-Implicit* pattern (Fig. 2d), a consumer C_i sends a request to a producer P_j. After processing C_i's request, P_j sends a reply to C_i and any other consumer C_k that may benefit from that reply. We identify two scenarios where this pattern could be used. In the first scenario, C_i is a wireless service (e.g., a nurse requesting a patient's lab results from a hospital server) and P_j broadcasts the reply via wireless channel. Another consumer C_k (e.g., the doctor in charge of the patient) listening on the channel may get the information if it finds it relevant [29]. In the second scenario, a department within a university may send a "request for further information" to the Dean's office about a recent regulation. The Dean's office may reply to that department and forward the reply to all departments within the college.

Push-based dissemination—Push-based dissemination includes four patterns (Fig. 3). In the *Push-1:1-Explicit* pattern (Fig. 3a), there is a one-to-one subscription relationship between consumers and producers. We identify three scenarios relevant to this pattern. The first scenario corresponds to the "Pipes and Filters" architectural pattern [10]. "Pipes and Filters" provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter service. Data is passed through pipes between adjacent filters. Each filter service gets the input data from the previous pipe, performs a function on the input data, and supplies the output to the next filter. The second scenario corresponds to multilayer architectures where interactions occur only between adjacent layers. The third scenario corresponds to a hierarchical publish-subscribe scheme. Services are organized in a tree and allowed to subscribe only with their

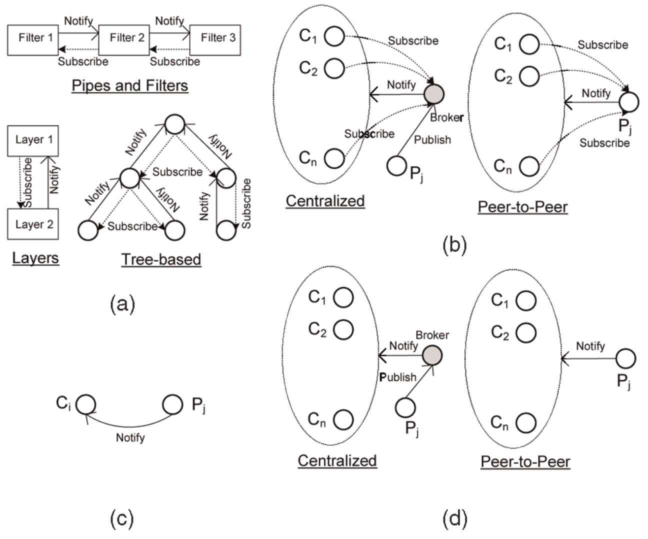


Fig. 3. Scenarios for push-based dissemination. (a) Pull-1:1-Explicit. (b) Pull-1:N-Explicit. (c) Pull-1:1-Implicit. (d) Pull-1:N-Implicit.

children. Therefore, notifications may go only from a service to its parent. This pattern is used, for instance, in organizations with hierarchically structured components, where each component reports only to its parent.

The *Push-1:N-Explicit* pattern (Fig. 3b) corresponds to the publish-subscribe interaction scheme. It uses either a centralized or peer-to-peer architecture. Details about this pattern are given in [9], [21], and [42]. The *Push-1:1-Implicit* pattern (Fig. 3c) corresponds to WSDL's notify operations [1], [36] where a Web service sends an output message to another service. No explicit subscription is required from the consumer to receive a notification. The service to be notified (i.e., consumer) may, for instance, be specified in the business logic of the notifier service (i.e., producer). The *Push-1:N-Implicit* (Fig. 3d) represents the case where a producer submits information to a set of relevant services without explicit subscriptions from those services. This pattern may adopt a centralized or peer-to-peer topology. It will be covered in details in Sections 3 and 4.

2.2 Filtering Taxonomy

Consumers are usually interested in particular events or event types, and not all the events. Filtering mechanisms are used to help specify/disseminate events of interest. The *filtering taxonomy* (Fig. 4) defines two dimensions for such mechanisms: *event* and *service* dimension.

Event dimension—This dimension deals with filtering at the event level (i.e., which event is relevant). It includes the following five cases: *topic-based*, *content-based*, *type-based*, *semantic*, and *generic*. In *topic-based* filtering, each event belongs to one of a fixed set of subjects (also called topics); producers are required to label each event with a topic name. Consumers subscribe to topics. Events of topic T will be sent to all consumers that subscribed to T. In *content-based* filtering, events are no longer divided into different subjects. Consumers define a subscription condition according to the internal structure of events (e.g., stock quotes that cost lest than \$100); all events that meet the condition will be sent to those consumers. In *type-based* filtering, events are

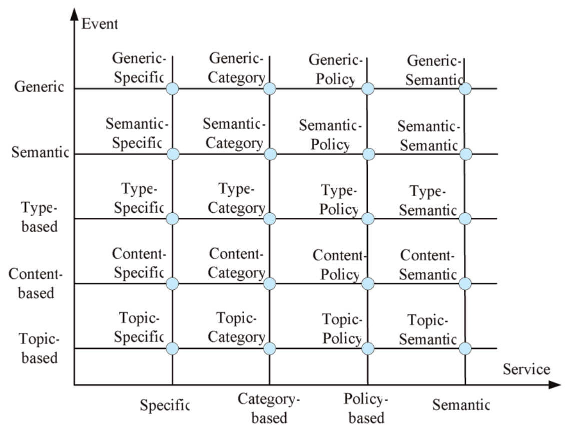


Fig. 4. Filtering taxonomy.

instances of application-defined types. For instance, WSDL (the standard for defining Web service interfaces) uses XML Schema as a typing system. Consumers are interested in receiving events that have a certain type. In the *semantic* event filtering, events are semantically described according to a domain ontology [17]. Filtering is based on a condition over the semantic properties of the events. In the *generic* event filtering, there is no condition on the event to be disseminated.

Service dimension—This dimension deals with filtering at the service level (consumers and producers). It includes the following four cases: *specific*, *category-based*, *policy-based*, and *semantic*. In the *specific* filtering, consumers receive events from specific producers they are registered with. In the *category-based* filtering, consumers receive events from producers that belong to certain categories. Each category defines a domain of interest (e.g., airline and healthcare) of Web services in the SOA. In the *policy-based* filtering, consumers receive events from producers that implement certain policies. We adopt a broad definition of policy [24], encompassing not only security and privacy but also the capabilities and requirements under which a service may interact with clients. For instance, a service may be interested in events published by services that implement a given WSDL interface. In the *semantic* filtering, consumers receive events from producers that satisfy certain semantic conditions (functional or nonfunctional). This assumes that all services are described according to a semantic model such as DAML-S [28], OWL-S [26], and WSMO [38].

3 The Implicit Notification Model

Our focus in the rest of this paper is on defining protocols for enabling the peer-to-peer Push-1:N-Implicit pattern with a topic-category-based filtering. For simplicity, we will use the term *implicit notification* to refer to this pattern. In the topic-category-based filtering, consumers receive events under certain topics and are published by producers that belong to certain categories.

In this section, we describe our implicit notification model. We first present the concept of implicit notification ontology (INO) to model interactions among Web services. Then, we propose a peer-to-peer topology for exchanging notifications among Web services. Finally, we introduce the Basic Implicit Notification (BIN) protocol.

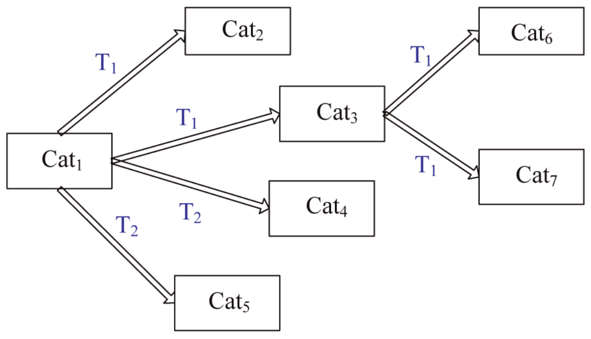


Fig. 5. Example of INO graph.

3.1 Implicit Notification Ontology

We model the different ways through which Web services exchange implicit notifications through the INO. Simply put, ontology is a formal and explicit specification of a shared conceptualization [5], [17], [28]. At an abstract level, we model INO as a labeled directed graph where nodes represent concepts in the application domain and labeled edges represent relationships between concepts. The INO graph can easily be specified using major ontology languages such as RDF and OWL [17]. Each concept in the graph refers to a service category (i.e., domain of interest) from the application domain. An edge $\mathrm{Cat_i} \rightarrow \mathrm{Cat_j}$ labeled with T means that services that belong to category $\mathrm{Cat_i}$ share information of topic T with services of category $\mathrm{Cat_j}$.

The categories (i.e., nodes) and topics (edges) of the INO graph are defined according to two categorizations: *topic* and *service categorization*. The topic categorization gives the various topics of messages that may be exchanged among Web services. For instance, let us consider disaster management as a case study. The topic categorization may include topics such as "disaster supplier info" and "press release." A message M is defined by the couple (T,D) where T is a topic and D is the actual data to be sent. Topics may be recursively organized into subtopics. We use the notation Subtopics(T) to refer to all subtopics under T (descendents). The service categorization gives the categories (domains of interest) of services that may need to exchange messages. For instance, the service categorization in a disaster management system may include categories such as "medicine food supplies" and "news media." A service may belong to more than one category. A category may be recursively organized into subcategories. We use the function Subcategories(C) to refer to all subcategories under C (descendents).

Fig. 5 gives an example of an INO graph. It depicts five service categories: $\mathrm{Cat_1}$, $\mathrm{Cat_2}$, $\mathrm{Cat_3}$, $\mathrm{Cat_4}$, and $\mathrm{Cat_5}$. It states that services of category $\mathrm{Cat_1}$ notify services of categories $\mathrm{Cat_2}$ and $\mathrm{Cat_3}$ about messages of topic $\mathrm{T_1}$ and services of categories $\mathrm{Cat_4}$ and $\mathrm{Cat_5}$ about messages of topic $\mathrm{T_2}$. Services of category $\mathrm{Cat_3}$ notify services of categories $\mathrm{Cat_6}$ and $\mathrm{Cat_7}$ about messages of topic $\mathrm{T_1}$.

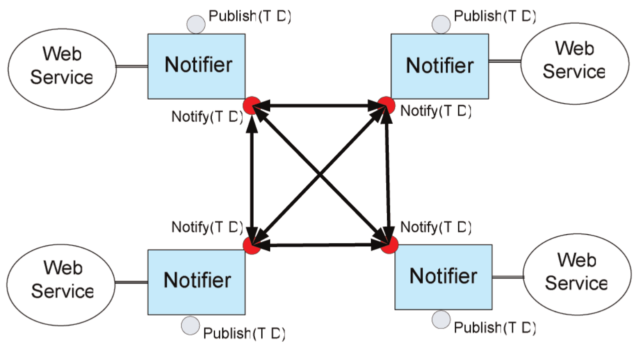


Fig. 6. Interactions among notifiers.

The INO graph allows the definition of a *neighborhood* relationship among Web services. This is a nonsymmetric relationship defined as follows: Let us consider two services WS_1 and WS_2 of categories Cat_1 and Cat_2, respectively. We say that WS_2 is a *neighbor* of WS_1 *with respect to* a topic T if the INO graph includes an edge $Cat_1 \rightarrow Cat_2$ labeled with T. Our definition of "neighbor" is different from the one used in gossip-based (or epidemics-style) protocols [8], [16]. First, our definition is logical (based on the ontology graph), while the one used in gossip protocols is physical (based on the network topology). Second, our definition depends on the topic of the message to be propagated. Finally, neighbors in our case are selected based on the INO graph and, hence, are necessarily interested in receiving the message. In gossip protocols, neighbors are randomly selected and, hence, may not be interested in receiving the message.

The INO graph is distributed over the different services in the system. If a service has Cat_i as a category, then it is only aware of the relationships with Cat_i as a source. For instance, a service of category Cat_1 (Fig. 5) knows that it should send messages of topic T_1 to services of categories Cat_2 and Cat_3. However, it is not aware that services of category Cat_3 will forward those messages to services of categories Cat_6 and Cat_7. This is similar to the Chain of Responsibility design pattern [19], where an object does not know the objects beyond its direct successor. For instance, in disaster management, each entity is only aware of the entities it should directly interact with (e.g., local-to-state-to-federal-to-international).

We identify two approaches for defining INO graphs: *manual* and *automatic.* Details about these approaches are out of the scope of this paper. For completeness, we give a definition of each approach in the following:

- *Manual.* The INO graph is manually created by domain experts. Experts in each domain or category Cat_i identify the part of INO related to Cat_i (i.e., relationships with Cat_i as a source). Domain experts are also responsible for maintaining the part of INO related to their category (e.g., changing topics or adding relationships).
- *Automatic.* The INO graph is (semi)automatically created and maintained. One solution is to use machine learning [31] for creating INO. For instance, the system can learn (via data mining) from the subscriptions made in push explicit techniques (Figs. 3a and 3b) to determine the relationships in INO. Another solution is to adopt a mass collaboration approach [27]. Mass collaboration has been employed quite successfully in open-source software (e.g., Linux), product reviews (e.g., amazon.com), and collaborative filtering. In our case, we propose applying it in conjunction with manual and automatic techniques to build the INO graph. The idea is to build an INO "shell" and then leverage the mass of users (consumers, producers, system administrators, etc.) to help populate the INO graph or verify and correct the relationships created by automatic methods.

TABLE 1
BIN Algorithm Executed by Notifier N_i

```
(1)  At Invocation of Publish(T,D) or Notify(T,D) {
(2)    NeighborSet = GetNeighbors(T,$N_i$)
(3)    For each $N_k \in$ NeighborSet Do
(4)      Invoke Notify(T,D) of $N_k$
(5)    EndFor }

(6)  Function GetNeighbors(T,N) Returns Set {
(7)    Determine Category $Cat_p$ of N
(8)    $\tau = \{T\} \cup$ Subtopics(T)
(9)    NotifierSet = $\emptyset$
(10)   For each link $Cat_p \rightarrow Cat_q$ labeled with a topic from $\tau$ Do
(11)     $\zeta = \{Cat_q\} \cup$ Subcategories($Cat_q$)
(12)     For each Notifier $N_j$ of a category from $\zeta$ Do
(13)       NotifierSet = NotifierSet $\cup \{N_j\}$
(14)     End For
(15)   End For
(16)   NotifierSet = NotifierSet - {N}
(17)   return NotifierSet }
```

3.2 Peer-to-Peer Topology for Implicit Notifications

One important feature of the proposed notification infrastructure is the automatic interaction among services to share topic-based messages. Each Web service has an administrative service, called *notifier*, associated to it (Fig. 6). All notifiers are registered in the service registry (UDDI in our case); they are categorized according to the type of service they are associated to (as defined in the service categorization).

Notifiers define a peer-to-peer network for sharing information among services. Each provider publishes information via its associated notifier. The notifier formats the published information into a message and sends it to other peers using the protocol described in Table 1. At the reception of a message, notifiers apply the same protocol to forward the message to other notifiers. The WSDL document of each notifier N_i corresponding to service WS_i includes two operations: Publish(T,D) and Notify(T,D). Publish(T,D) allows provider-to-notifier interactions. It is invoked by the WS_i provider to disseminate the message M(T,D) through N_i. We say that N_i is the *root notifier* of M. Notify(T,D) allows notifier-to-notifier interactions. It enables the exchange of M(T,D) among notifiers. In the rest of this paper, we will use a service and its attached notifier interchangeably.

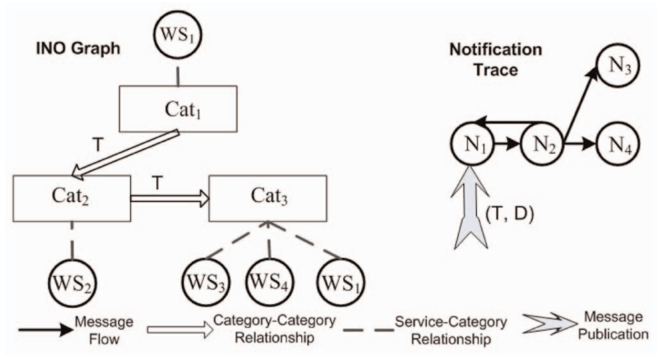


Fig. 7. Example of notification loops in BIN.

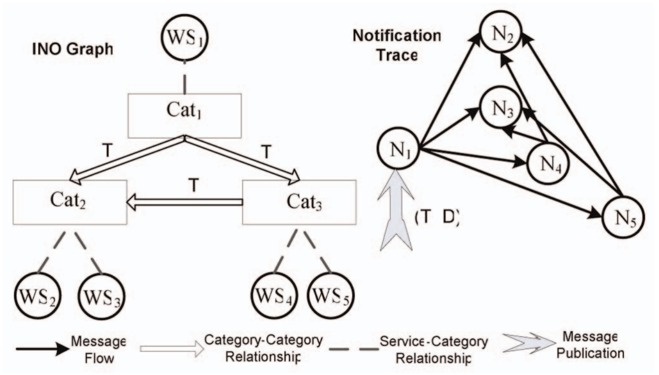


Fig. 8. Example of double notifications in BIN.

The proposed notification topology is attractive for a number of reasons. First, it distributes control over several architectural modules. Every service acts both as a consumer and producer, thus precluding specific brokers which would prevent scalability and fault-tolerance [2]. Second, it externalizes notification management and creates a clear separation between the business logic of the service and dissemination tasks. Third, it facilitates the integration of notification mechanisms in legacy SOAs (i.e., SOAs designed with no notification capabilities). Fourth, it preserves the autonomy of Web services since little or no modification needs to be done to Web services to send notifications.

3.3 Basic Implicit Notification Protocol

In this section, we present the BIN protocol. BIN uses the INO graph and adopts the peer-to-peer topology (Fig. 6) for disseminating information among Web services. Table 1 gives the BIN protocol executed by notifier $\mathrm{N_i}$. In this protocol, both Publish() and Notify() operations are defined in the same way. At the invocation of Publish(T,D) or Notify(T,D), $\mathrm{N_i}$ calls the $\mathrm{GetNeighbors(T, N_i)}$ to determine the set of neighbors. Then, $\mathrm{N_i}$ forward M(T,D) to each notifier $\mathrm{N_k}$ in NeighborSet by invoking the operation Notify(T,D) of $\mathrm{N_k}$.

The $\mathrm{GetNeighbors(T, N_i)}$ function uses the INO graph to determine neighbors. Let us assume that $\mathrm{Cat_p}$ is the category of $\mathrm{N_i}$. $\mathrm{N_i}$ first gets all edges $\mathrm{Cat_p} \rightarrow \mathrm{Cat_q}$ labeled with T or a subtopic of T from the INO graph. Then, $\mathrm{N_i}$ inserts each notifier $\mathrm{N_k}$ with $\mathrm{Cat_q}$ or a subcategory of $\mathrm{Cat_q}$ as a category in the NotifierSet. If $\mathrm{N_i}$ belongs to NotifierSet, then $\mathrm{N_i}$ deletes itself from NotifierSet. This may happen if 1) the INO graph has a loop on $\mathrm{Cat_p}$ or 2) INO has an edge $\mathrm{Cat_p} \rightarrow \mathrm{Cat_q}$ labeled with T, and $\mathrm{N_i}$ belongs to categories $\mathrm{C_p}$ and $\mathrm{C_q}$. Finally, the GetNeighbors() function returns NotifierSet.

BIN has two major drawbacks. First, a message may be sent indefinitely in the system (notification loops). Second, a message may be received repeatedly by a notifier (double notifications).

Notification loops in BIN. Fig. 7 gives an example of notification loop. We assume that $\mathrm{WS_1}$ is registered under $\mathrm{Cat_1}$ and $\mathrm{Cat_3}$, $\mathrm{WS_2}$ is registered under $\mathrm{Cat_2}$, and $\mathrm{WS_3}$ and $\mathrm{WS_4}$ are registered under $\mathrm{Cat_3}$. Let us assume that M(T,D) is published to $\mathrm{N_1}$. Based on the INO graph, $\mathrm{N_1}$ (under $\mathrm{Cat_1}$) sends M to $\mathrm{N_2}$ (under $\mathrm{Cat_2}$). Then, $\mathrm{N_2}$ sends M to services under $\mathrm{Cat_3}$ (i.e., $\mathrm{N_3}$, $\mathrm{N_4}$, and $\mathrm{N_1}$). $\mathrm{N_1}$ will again forward M to $\mathrm{N_2}$, hence creating an infinite notification loop.

Double notifications in BIN. Fig. 8 gives an example of double notification (or repetition). $\mathrm{WS_1}$ is registered under $\mathrm{Cat_1}$, $\mathrm{WS_2}$ and $\mathrm{WS_3}$ are registered under $\mathrm{Cat_2}$, and $\mathrm{WS_4}$ and $\mathrm{WS_5}$ are registered under $\mathrm{Cat_3}$. Let us assume that M(T,D) is published to $\mathrm{N_1}$. Based on the INO graph, $\mathrm{N_1}$ sends M to $\mathrm{N_2}$, $\mathrm{N_3}$, $\mathrm{N_4}$, and $\mathrm{N_5}$. At the reception of M, $\mathrm{N_4}$ and $\mathrm{N_5}$ forward M to $\mathrm{N_2}$ and $\mathrm{N_3}$ leading to double notifications.

One solution for detecting notification loops and repetitions is by checking the INO graph. However, such a solution is not practical for the following reasons: First, the INO graph is distributed over the different services in the system. Each service is only aware of the part of the graph related to its category. No one has a global view on INO. Second, even if there is no cycle in the INO graph, there is still a risk of notification loops because a service may belong to several categories (see the example in Fig. 7). Third, Web services operate in a dynamic environment; they may join/leave the service space at any time. Additionally, topics, categories, and relationships among categories may be added/changed/removed at any time. Hence, there is a need to extend BIN to handle notification loops and repetitions.

4 Implicit Notification Protocols

In this section, we propose three implicit notification protocols to deal with repetitions and notification loops: Centralized Implicit Notification (CIN), Distributed Implicit Notifications (DIN), and Header-based Implicit Notifications (HIN).

4.1 Centralized Implicit Notification

The CIN protocol uses a centralized history to keep track of all messages along with their consumers/producers. The history can be seen as a table with three columns: ID, Notifier, and Type. An entry (ID,N,Type) in the history means that notifier N processed the message identified by ID as a consumer $(\mathrm{Type} = 1)$ or producer $(\mathrm{Type} = 2)$. We use Universal Unique Identifiers (UUIDs) as message IDs; UUIDs are 128-bit numbers used to uniquely identify an object or entity on the Internet. IDs are carried by messages throughout the notification process. We assume the existence of a function GenerateID() that generates UUIDs.

Table 2 gives the CIN algorithm executed by $\mathrm{N_i}$. At the invocation of Publish(T,D), $\mathrm{N_i}$ (the root notifier) generates a

TABLE 2
CIN Algorithm Executed by Notifier N_i

```
(1)  At Invocation of Publish(T,D)  {
(2)    ID = GenerateID()
(3)    Insert(ID,$N_i$,2) in History
(4)    NeighborSet = GetNeighbors(T,$N_i$)
(5)    For each $N_k \in$ NeighborSet Do
(6)      Insert(ID,$N_k$,1) in History
(7)      Invoke Notify(ID,T,D) of $N_k$
(8)    EndFor  }

(9)  At Invocation of Notify(ID,T,D)  {
(10)  If (ID,$N_i$,2) $\notin$ History
(11)    Then Insert(ID,$N_i$,2) in History
(12)         NeighborSet = GetNeighbors(T,$N_i$)
(13)         For each $N_k \in$ NeighborSet Do
(14)         If (ID,$N_k$,*) $\notin$ History
(15)           Then  Insert(ID,$N_k$,1) in History
(16)                 Invoke Notify(ID,T,D) of $N_k$
(17)         EndIf
(18)  EndIf }
```

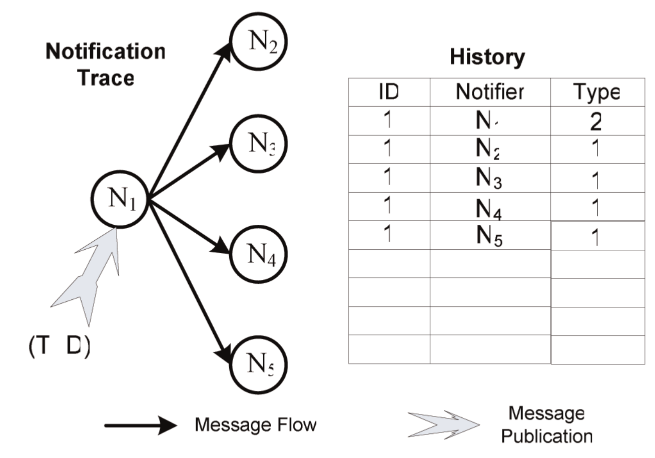
unique message ID. Since N_i is the producer of M(T,D), it stores $(ID, N_i, 2)$ in the history (N_i is a consumer and will act as producer by forwarding M to neighbors). N_i then determines the list of neighbors. Finally, for each notifier N_k that belongs to NeighborSet, N_i inserts $(ID, N_k, 1)$ in the history and forwards M(ID,T,D) to N_k.

At the invocation of Notify(ID,T,D), N_i first checks whether it already processed the message as a producer, i.e., if $(ID, N_i, 2) \in History$. This may happen if two or more producers check the history at the *same* time, do not find $(ID, N_i, ^*)$ in the history, and, hence, they all decide to send M to N_i. In this case, N_i processes the first received message only. If $(ID, N_i, 2) \notin History$, N_i first inserts $(ID, N_i, 2)$ in the History. Then, it computes the NeighborSet. Let N_k be a notifier in NeighborSet. N_i forwards M(ID,T,D) to N_k only if N_k has not previously received this message; this is done by checking that $(ID, N_k, ^*)$ does not belong to the history. If $(ID, N_k, ^*) \notin History$, N_i inserts $(ID, N_k, 1)$ in the history and sends M to N_k. Otherwise, N_i ignores N_k.

Lemma 1. *No notifier forwards a message M(ID,T,D) twice to the same neighbor in the CIN protocol.*

Proof. Let us assume that a notifier N_i sends a message M(ID,T,D) to the same neighbor N_j at two different times t_1 and t_2 ($t_1 < t_2$). We consider the following cases:

- N_i is a root notifier: N_i receives the message via Publish(). According to the CIN protocol (line 3), N_i inserts $(ID, N_i, 2)$ in History at t_1. At time t_2, N_i first checks whether $(ID, N_i, 2)$ belongs to History (line 10) before sending M. Hence, N_i cannot forward M to N_j at t_2 since $(ID, N_i, 2)$ is already in History.
- N_i is not a root notifier: N_i is the first to receive the message via Notify(). Hence, $(ID, N_i, 2) \notin History$. According to the CIN protocol (line 11), N_i inserts $(ID, N_i, 2)$ in History at t_1. At time t_2, N_i first checks whether $(ID, N_i, 2)$ belongs to History (line 10) before sending M. Hence, N_i cannot forward M to N_j at t_2 since $(ID, N_i, 2)$ is already in History. □



ID	Notifier	Type
1	N_1	2
1	N_2	1
1	N_3	1
1	N_4	1
1	N_5	1

Fig. 9. CIN protocol—An example.

To illustrate the CIN protocol, let us consider the scenario depicted in Fig. 9 that corresponds to the INO graph given in Fig. 8. Let us assume that (T,D) is published to N_1. N_1 generates a unique message ID equal to 1 and inserts $(1, N_1, 2)$ in the history. Based on the INO graph, N_1 sends M(1,T,D) to N_2, N_3, N_4, and N_5. It also inserts $(1, N_2, 1)$, $(1, N_3, 1)$, $(1, N_4, 1)$, and $(1, N_5, 1)$ in the History. At the reception of M(1,T,D), N_4 and N_5 figure out that N_2 and N_3 are neighbors. Since $(1, N_2, 1)$ and $(1, N_3, 1)$ belong to the history, N_4 and N_5 do not forward M to N_1 and N_2. The total number of notifications sent is 4 (no repetitions).

4.2 Distributed Implicit Notification

The use of a centralized history in CIN suffers from two major drawbacks. First, the history node constitutes a single point of failure. Second, each notifier needs to remotely access the history, which may increase the time required to send notifications. To address these problems, we propose a distributed version of CIN. We refer to this protocol as DIN.

In this protocol (Table 3), each notifier N_i maintains a local history called $History_i$. $History_i$ contains the IDs of all messages received by N_i (as a producer or consumer). If

TABLE 3
DIN Algorithm Executed by Notifier N_i

```
(1)  At Invocation of Publish(T,D)  {
(2)    ID = GenerateID()
(3)    Insert(ID) in $History_i$
(4)    NeighborSet = GetNeighbors(T,$N_i$)
(5)    For each $N_k \in$ NeighborSet Do
(6)      Invoke Notify(ID,T,D) of $N_k$
(7)    EndFor  }

(8)  At Invocation of Notify(ID,T,D)  {
(9)    If ID $\notin$ $History_i$
(10)     Then Insert(ID) in $History_i$
(11)          NeighborSet = GetNeighbors(T,$N_i$)
(12)          For each $N_k \in$ NeighborSet Do
(13)              Invoke Notify(ID,T,D) of $N_k$
(14)          EndFor
(15)   EndIf }
```

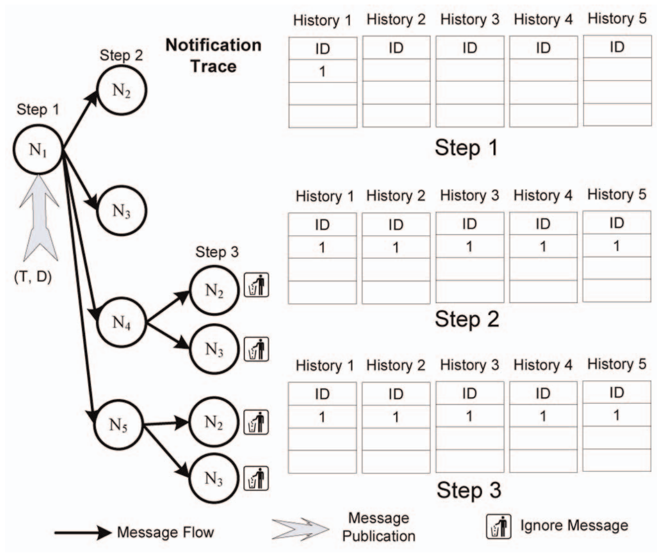


Fig. 10. DIN protocol—An example.

the operation invoked is Publish(T,D), N_i generates a unique message ID, inserts ID in $History_i$, and computes NeighborSet. Finally, N_i forwards M(ID,T,D) to each notifier N_k in NeighborSet. If the operation invoked is Notify(ID,T,D), N_i checks if ID belongs to $History_i$. If so, N_i ignores M since it has already received the message. Otherwise, N_i inserts ID in $History_i$, computes NeighborSet, and forwards M to each notifier N_k in NeighborSet.

Lemma 2. *No notifier forwards a message M(ID,T,D) twice to the same neighbor in DIN protocol.*

Proof. The proof is similar to the one given for Lemma 1 (replace History and $(ID, N_i, 2)$ by $History_i$ and ID, respectively). □

To illustrate the DIN protocol, let us consider the scenario depicted in Fig. 10 that corresponds to the INO graph given in Fig. 8. This scenario shows that repeated notifications are still possible, but detected, in DIN. Since N_1 is the first to receive (T,D), it generates a unique ID (say 1), inserts ID in $History_1$, and forwards M(1,T,D) to notifiers that belong to Cat_2 and Cat_3 (i.e., N_2, N_3, N_4, and N_5). Let us assume that N_2, N_3, N_4, and N_5 receive M at the same time. At the reception of M, N_2, N_3, N_4, and N_5 insert ID into their local histories since ID does not belong to any of those histories. Based on the INO graph, N_4 and N_5 forward M to notifiers that belong to Cat_2 (i.e., N_2 and N_3). Since ID belongs to $History_2$ and $History_3$, N_2 and N_3 consider M as a repeated notification and, hence, ignore the message. The total number of notifications sent is eight, including four repetitions.

4.3 Header-Based Implicit Notification

The HIN protocol uses the SOAP header of each notification message to store the names of notifiers that already received the message. When a notifier receives a message, it decides whether to forward the message to the next notifier by simply looking at the header of that message. The header of a message M(T,D) is composed of two parts: *Root* and *Notified*. *Root* contains the name of the root notifier. *Notified* is a list of all notifiers (except the root) that received M.

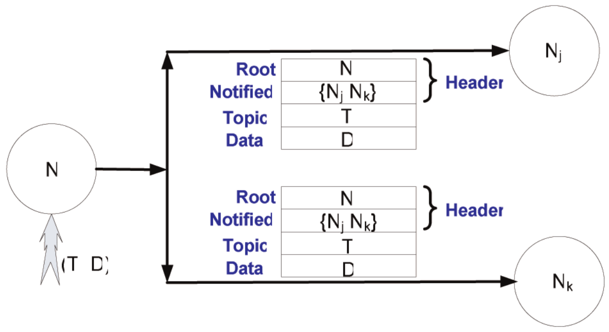


Fig. 11. Structure of a notification message in HIN.

Fig. 11 shows an example of notification message in HIN, where M is published to the root notifier N_i. Let us assume that N_i needs to send M to N_j and N_k. N_i sets M.Header.Root to N_i, M.Header.Notified to $\{N_j, N_k\}$, and sends M(Header,T,D) to N_j and N_k.

Table 4 gives the HIN algorithm executed by a notifier N_i. At the invocation of Publish(T,D), N_i creates a Header and assigns N_i to Header.Root. Next, N_i determines the list of neighbors. Finally, N_i assigns NeighborSet to M.Notified and sends M(Header,T,D) to each notifier in NeighborSet. At the invocation of Notify(Header,T,D), N_i determines the list of neighbors. If N_i belongs to NeighborSet, then N_i deletes itself from NeighborSet. Then, N_i considers the following two cases for each notifier N_k in NeighborSet:

1. $N_k = Header.Root$ or $N_k \in Header.Notified$. N_k already received the message. Hence, N_i ignores N_k and removes N_k from NeighborSet.
2. $N_k \neq Header.Root$ and $N_k \notin Header.Notified$. N_k did not receive the message before. Hence, N_i adds N_k to Header.Notified.

TABLE 4
HIN Algorithm Executed by Notifier N_i

```
(1)  At Invocation of Publish(T,D) {
(2)   Create Header
(3)   M.Header.Root = $N_i$
(4)   NeighborSet = GetNeighbors(T,$N_i$)
(5)   M.Header.Notified = NeighborSet
(6)   For each $N_k \in$ NeighborSet Do
(7)     Invoke Notify(Header,T,D) of $N_k$
(8)   EndFor }

(9)  At Invocation of Notify(Header,T,D) {
(10)  NeighborSet = GetNeighbors(T,$N_i$)
(11)  For each $N_k \in$ NeighborSet Do
(12)     If $N_k$ = Header.Root $\vee$ $N_k \in$ Header.Notified
(13)          Then NeighborSet = NeighborSet - {$N_k$}
(14)          Else Header.Notified = Header.Notified $\cup$ {$N_k$}
(15)     EndIf
(16)  EndFor
(17)  For each $N_k \in$ NeighborSet Do
(18)    Invoke Notify(Header,T,D) of $N_k$
(19)  EndFor }
```

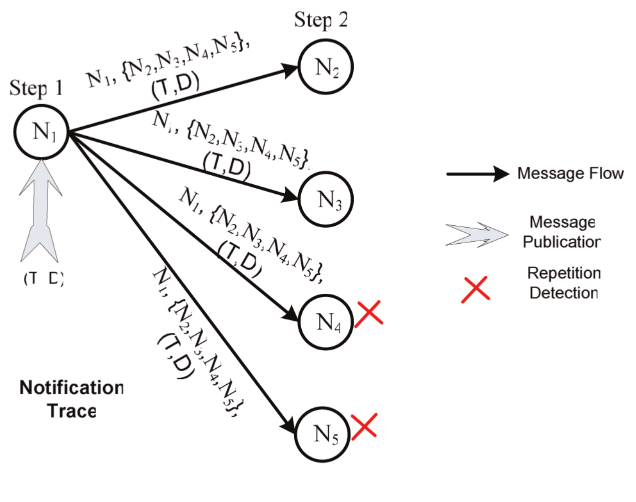


Fig. 12. HIN protocol—An example.

Finally, N_i forwards M(Header,T,D) to each notifier N_k that belongs to NeighborSet by invoking the operation N_k.Notify(Header,ID,T,D).

Lemma 3. *No notifier forwards a message M(Header,T,D) twice to the same neighbor in HIN protocol.*

Proof. Let us assume that a notifier N_i sends a message M(Header,T,D) to the same neighbor N_j at two different times t_1 and t_2 ($t_1 < t_2$). We consider the following cases:

- N_i is a root notifier: N_i receives the message via Publish(). According to HIN protocol (lines 3 and 5), N_i assigns N_i to M.Header.Root and N_j to M.Header.Notified at t_1. At time t_2, N_i removes N_j from NeighborSet since N_j is already in M.Header.Notified (lines 12 and 13). Hence, N_i does not forward M to N_j at t_2.
- N_i is not a root notifier: N_i is the first to receive the message via Notify(). According to the HIN protocol (line 14), N_i assigns N_j to M.Header.Notified at t_1. At time t_2, N_i removes N_j from NeighborSet since N_j is already in M.Header.Notified (lines 12 and 13). Hence, N_i does not forward M to N_j at t_2. □

Let us consider the scenario depicted in Fig. 12 that corresponds to the INO graph given in Fig. 8. Since N_1 is the root notifier, it initializes M.Header.Root with N_1. From the INO graph, N_3 determines that N_2, N_3, N_4, and N_5 are neighbors. Hence, N_3 initializes M.Header.Notified with $\{N_2, N_3, N_4, N_5\}$ and forwards M with the updated header to N_2, N_3, N_4, and N_5. N_4 and N_5 determine that services under category Cat_2 (i.e., N_2 and N_3) are neighbors. However, both N_2 and N_3 belong to M.Header.Notified. Therefore, N_4 and N_5 ignore M. The total number of notifications is four (no repetitions).

Although Fig. 12 shows that N_4 and N_5 are able to detect repetitions, double notifications are still possible in HIN. Let us consider the scenario shown in Fig. 13. We assume that N_1 is the root notifier. N_1 initializes M.Header.Root with N_1, assigns $\{N_2, N_3\}$ to M.Header.Notified, and forwards the message M with the updated header to N_2 and N_3. N_2 and N_3 determine that services under

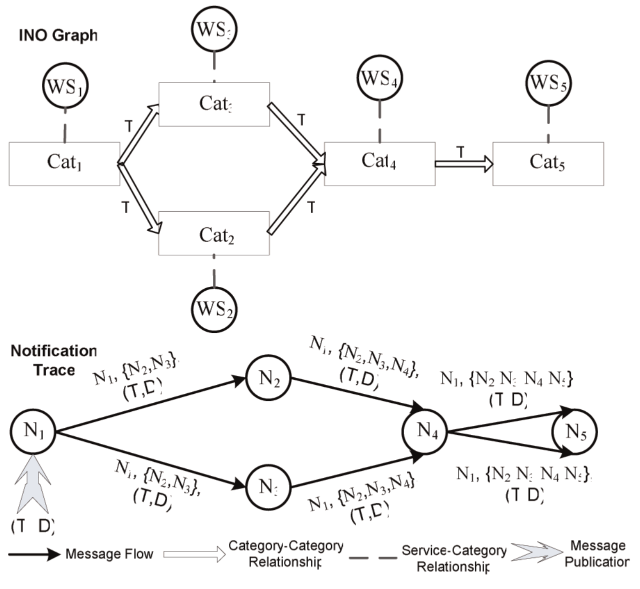


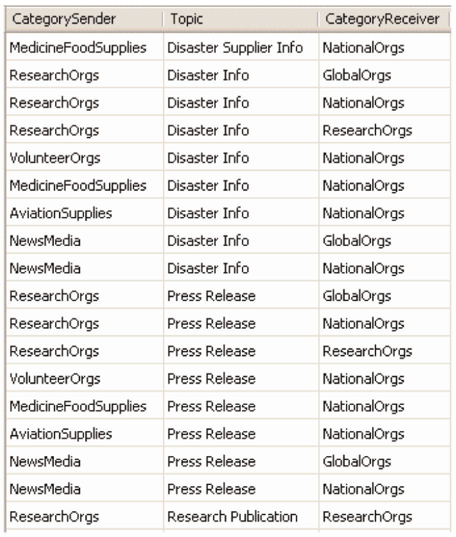
Fig. 13. Double notifications in HIN protocol.

category Cat_4 (i.e., N_4) are neighbors. However, N_4 is different from M.Header.Root and does not belong to M.Header.Notified. Hence, both N_2 and N_3 send M to N_4 (i.e., double notification). Since N_4 cannot determine whether it received the same message more than once, it sends M twice to N_5 (under category Cat_5). One solution to deal with this issue is to combine HIN with DIN; indeed, the use of local histories enables N_4 to keep track of previously received messages. The total number of notifications sent in Fig. 13 is six, including two repetitions. The same scenario executed with DIN would generate five notifications, including one repetition.

5 Implementation

In this section, we describe a prototype implementation of a disaster management notification system. We used Microsoft Windows Server 2003 (operating system), Microsoft Visual Studio 8 (development kit), UDDI Server, IIS Server, and SQL Server (for history tables in CIN and DIN). We defined the topic and service categorizations as UDDI categorizations. Fig. 14 shows part of the INO graph for disaster recovery implemented as a database table. The service categorization includes MedecineFoodSupplies, ResearchOrgs, NationalOrgs, VolunteerOrgs, GlobalOrgs, NewsMedia, and AviatioSupplies. The topic categorization includes disaster info, disaster supplier info, press release, and research publication.

As a proof of concept, we deployed 20 notifiers (implemented in C#) and registered them in UDDI under the corresponding categories. All notifiers are deployed in the same machine (Intel(R) processor, 1,500 MHz, and 512 Mbytes of RAM). The centralized history (in CIN) is stored in a separate machine within our local area network to simulate delays in accessing a remotely located history. The prototype includes graphical interfaces to automatically add notifiers to the system. Fig. 15 (top-left) depicts the screenshot for creating a new notifier named DetroitTV

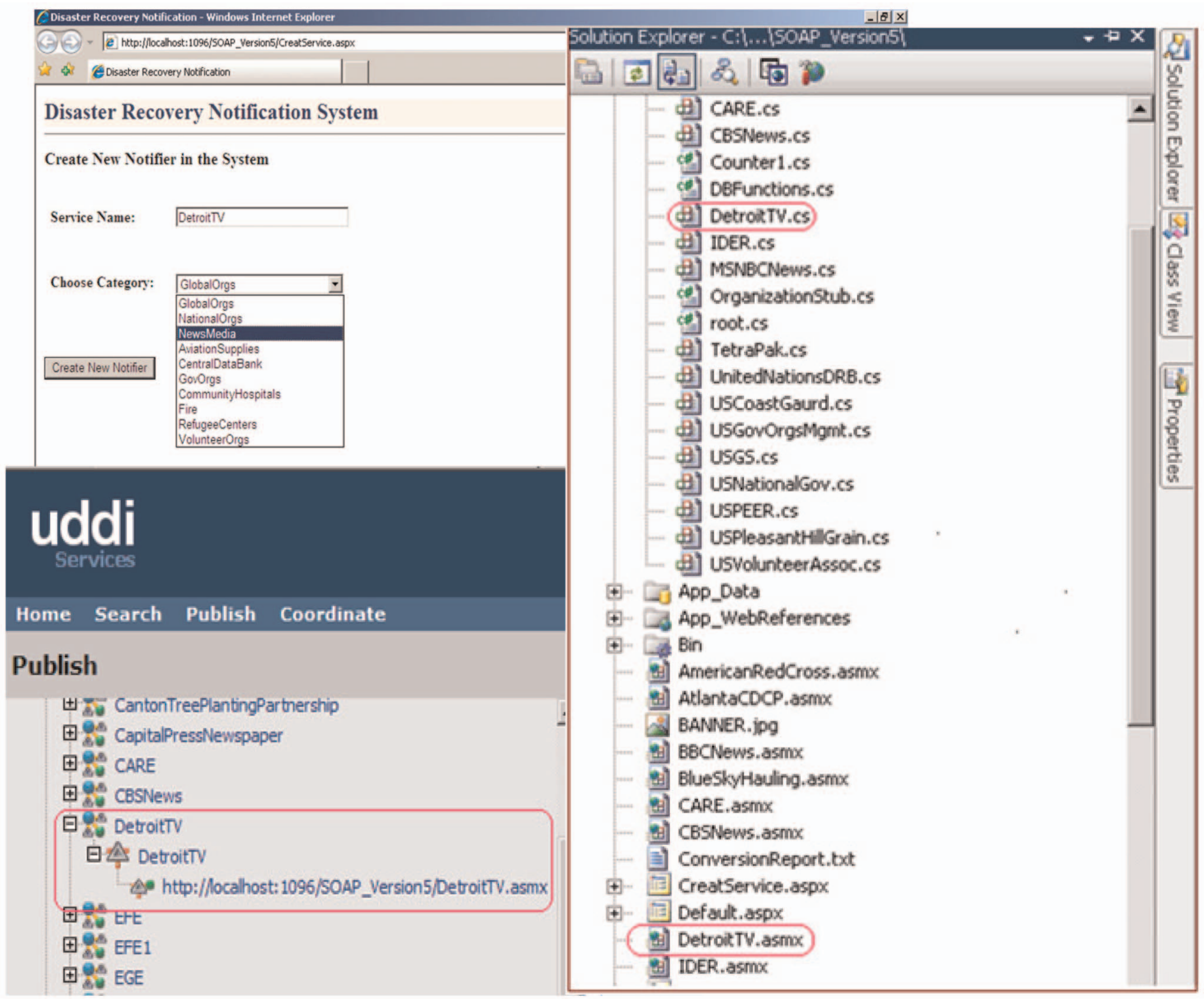


CategorySender	Topic	CategoryReceiver
MedicineFoodSupplies	Disaster Supplier Info	NationalOrgs
ResearchOrgs	Disaster Info	GlobalOrgs
ResearchOrgs	Disaster Info	NationalOrgs
ResearchOrgs	Disaster Info	ResearchOrgs
VolunteerOrgs	Disaster Info	NationalOrgs
MedicineFoodSupplies	Disaster Info	NationalOrgs
AviationSupplies	Disaster Info	NationalOrgs
NewsMedia	Disaster Info	GlobalOrgs
NewsMedia	Disaster Info	NationalOrgs
ResearchOrgs	Press Release	GlobalOrgs
ResearchOrgs	Press Release	NationalOrgs
ResearchOrgs	Press Release	ResearchOrgs
VolunteerOrgs	Press Release	NationalOrgs
MedicineFoodSupplies	Press Release	NationalOrgs
AviationSupplies	Press Release	NationalOrgs
NewsMedia	Press Release	GlobalOrgs
NewsMedia	Press Release	NationalOrgs
ResearchOrgs	Research Publication	ResearchOrgs

Fig. 14. INO graph for disaster recovery as a table.

under the category NewsMedia. The system automatically generates a "DetroitTV.cs" class and "DetroitTV.asmx" file (Fig. 15, top-right). It also automatically registers DetroitTV notifier under NewsMedia category in UDDI along with the access point for invoking the notifier (Fig. 15, bottom-left).

Fig. 16 depicts screenshots for publishing messages using the proposed protocols. The default page (Fig. 16, top-right) includes a text description of the system and prompts users to select the protocol they are interested in using (CIN, DIN, or HIN). Users are then taken to the next page (Fig. 16, middle) to select the topic, the root notifier, and the actual data to be published. In our example, the message "Earthquake occurred in California" with topic "Disaster Info" is published by International Disaster and Emergency Readiness (IDER). IDER belongs to the ResearchOrgs category. Based on the INO graph in Fig. 14, the message is forwarded to services under categories GlobOrgs (CARE and UnitedNationsDRB), NationalOrgs (USNationalGov), and ResearchOrgs (USGS and USPEER). Clicking on the Send button will invoke the Publish() method of the root notifier. Users will then be taken to the


Disaster Recovery Notification - Windows Internet Explorer
http://localhost:1096/SOAP_Version5/CreatService.aspx
Disaster Recovery Notification
Solution Explorer - C:\...\SOAP_Version5\
Solution Explorer
Class View
Properties
uddi
Services
Home
Search
Publish
Coordinate

Disaster Recovery Notification System

Create New Notifier in the System

Service Name: DetroitTV

Choose Category: GlobalOrgs

GlobalOrgs
NationalOrgs
NewsMedia
AviationSupplies
CentralDataBank
GovOrgs
CommunityHospitals
Fire
RefugeeCenters
VolunteerOrgs

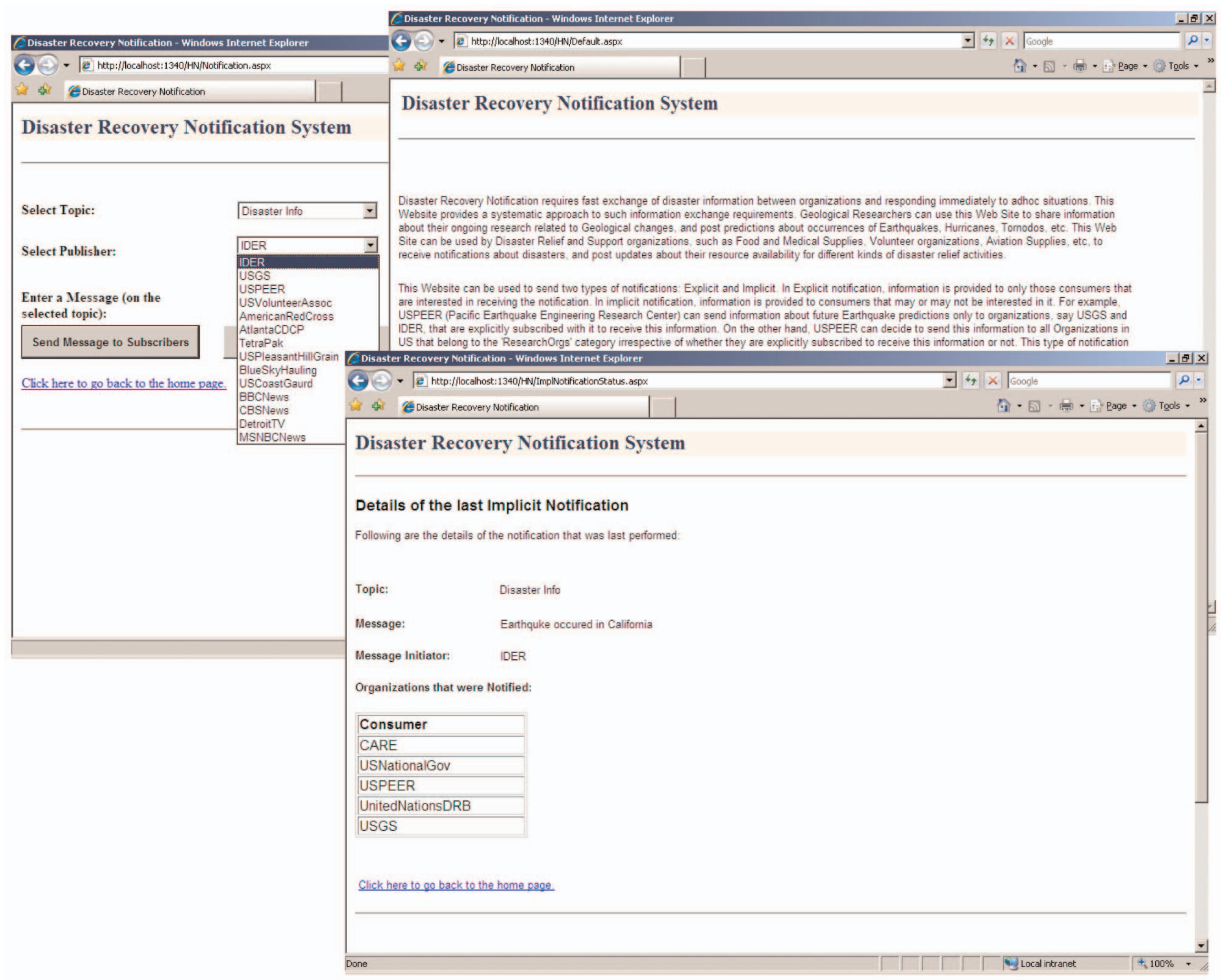
Create New Notifier

Publish

CantonTreePlantingPartnership
CapitalPressNewspaper
CARE
CBSNews
DetroitTV
DetroitTV
http://localhost:1096/SOAP_Version5/DetroitTV.asmx
EFE
EFE1
EGE

CARE.cs
CBSNews.cs
Counter1.cs
DBFunctions.cs
DetroitTV.cs
IDER.cs
MSNBCNews.cs
OrganizationStub.cs
root.cs
TetraPak.cs
UnitedNationsDRB.cs
USCoastGaurd.cs
USGovOrgsMgmt.cs
USGS.cs
USNationalGov.cs
USPEER.cs
USPleasantHillGrain.cs
USVolunteerAssoc.cs
App_Data
App_WebReferences
Bin
AmericanRedCross.asmx
AtlantaCDCP.asmx
BANNER.jpg
BBCNews.asmx
BlueSkyHauling.asmx
CARE.asmx
CBSNews.asmx
ConversionReport.txt
CreatService.aspx
Default.aspx
DetroitTV.asmx
IDER.asmx

Fig. 15. Prototype—Generating notifiers.


Disaster Recovery Notification - Windows Internet Explorer
http://localhost:1340/HN/Notification.aspx
Disaster Recovery Notification
http://localhost:1340/HN/Default.aspx
http://localhost:1340/HN/ImplNotificationStatus.aspx
Google
Page
Tools
Done
Local intranet
100%

Disaster Recovery Notification System

Select Topic: Disaster Info

Select Publisher: IDER

IDER
USGS
USPEER
USVolunteerAssoc
AmericanRedCross
AtlantaCDCP
TetraPak
USPleasantHillGrain
BlueSkyHauling
USCoastGaurd
BBCNews
CBSNews
DetroitTV
MSNBCNews

Enter a Message (on the selected topic):

Send Message to Subscribers

Click here to go back to the home page.

Disaster Recovery Notification System

Disaster Recovery Notification requires fast exchange of disaster information between organizations and responding immediately to adhoc situations. This Website provides a systematic approach to such information exchange requirements. Geological Researchers can use this Web Site to share information about their ongoing research related to Geological changes, and post predictions about occurrences of Earthquakes, Hurricanes, Tornodos, etc. This Web Site can be used by Disaster Relief and Support organizations, such as Food and Medical Supplies, Volunteer organizations, Aviation Supplies, etc, to receive notifications about disasters, and post updates about their resource availability for different kinds of disaster relief activities.

This Website can be used to send two types of notifications: Explicit and Implicit. In Explicit notification, information is provided to only those consumers that are interested in receiving the notification. In implicit notification, information is provided to consumers that may or may not be interested in it. For example, USPEER (Pacific Earthquake Engineering Research Center) can send information about future Earthquake predictions only to organizations, say USGS and IDER, that are explicitly subscribed with it to receive this information. On the other hand, USPEER can decide to send this information to all Organizations in US that belong to the 'ResearchOrgs' category irrespective of whether they are explicitly subscribed to receive this information or not. This type of notification

Disaster Recovery Notification System

Details of the last Implicit Notification

Following are the details of the notification that was last performed:

Topic: Disaster Info

Message: Earthquke occured in California

Message Initiator: IDER

Organizations that were Notified:

Consumer
CARE
USNationalGov
USPEER
UnitedNationsDRB
USGS

Click here to go back to the home page.

Fig. 16. Prototype—Sending implicit notifications.

result page (Fig. 16, bottom). This page displays the message ID generated by the root notifier, the topic and actual data of the message, and the list of services that have been automatically notified.

6 Experiments

We conducted experiments to compare the performance of CIN, DIN, and HIN protocols. We designed 10 scenarios in disaster management and ran each scenario through our prototype using CIN, DIN, and HIN protocols. Each scenario was design to include possibilities for double notifications, notification loops, or both. We measured the following two parameters:

1. *Number of notifications.* This parameter gives the number of times that a published message is sent to notifiers (the same or different notifiers). It is used to determine which protocol reduces the number of double notifications.
2. *Notification time.* This parameter gives the time (in seconds) it takes for a published message to reach all services interested in that message in the system. It is used to determine which protocol propagates a message faster in the system.

Fig. 17 compares the number of notifications generated in CIN, DIN, and HIN protocols. Experiments show that CIN always generates fewer notifications than DIN. As mentioned earlier, the use of a centralized history in CIN reduces double notifications compared to DIN as all notifiers have the same global view on the notification trace. Experiments show that HIN generates the same number of notifications as CIN in certain scenarios (e.g., scenarios 1, 4, 6, and 9). The notification trace in Fig. 12 is an

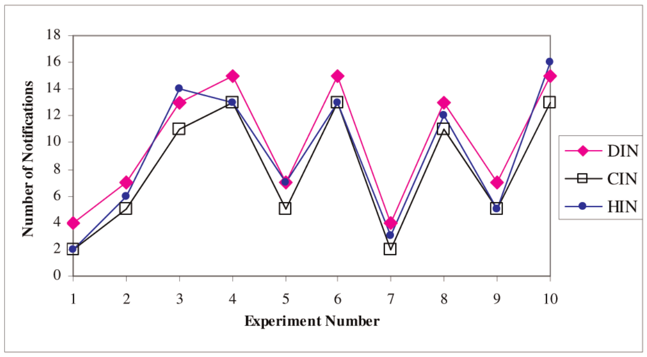


Fig. 17. Number of notifications in CIN, DIN, and HIN.

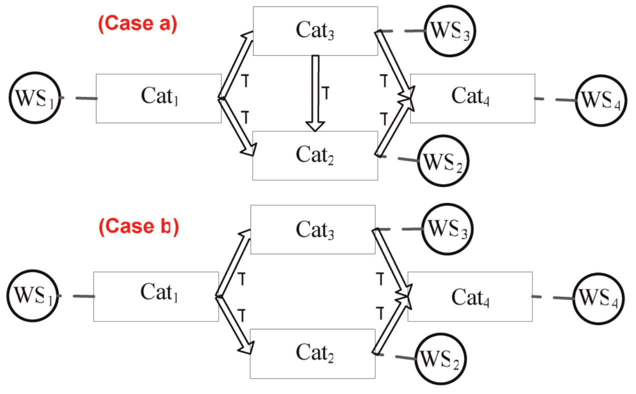


Fig. 18. Comparing DIN and HIN: Two cases.

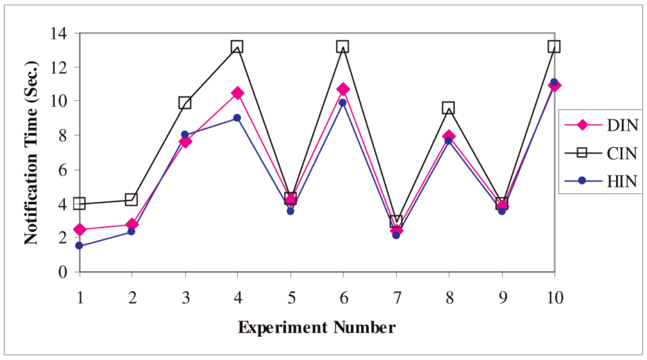


Fig. 19. Notification time in CIN, DIN, and HIN.

instance of such a situation. There are scenarios where DIN outperforms HIN (e.g., Scenarios 3 and 10). The notification trace in Fig. 13 is an instance of such a situation.

There are also scenarios where HIN is not as good as CIN, but it outperforms DIN (e.g., Scenarios 2, 7, and 8). For instance, in Fig. 18 (Case a), we assume that a message M of topic T is published to N_1. In this case, CIN generates three notifications, HIN generates four, and DIN generates five. Finally, there are situations where HIN and DIN generate the same number of notifications (e.g., Scenario 5). For instance, in Fig. 18 (Case b), we assume that a message M of topic T is published to N_1. In this case, CIN generates three notifications while both HIN and DIN generate four.

Fig. 19 compares the notification times of CIN, DIN, and HIN protocols. The notification time is bigger in the case of CIN since each notifier needs to access a history table located in a separate node before sending notifications. In DIN, each notifier accesses a local history table which reduces the overall notification time compared to CIN. Fig. 19 shows that DIN and HIN have almost similar notification times, with HIN performing slightly better in certain scenarios (e.g., scenarios 1, 4, and 6). This is because HIN generates much fewer notifications than DIN in those scenarios. Additionally, HIN does not access any database table to determine whether a notifier should forward a message or not.

7 Related Work

The Web has become the medium of choice for disseminating fast-changing data such as traffic/weather information, stock prices, and sports scores [6]. The growing popularity of RSS feeds and similar technologies illustrated the importance of this medium. RSS is a format for delivering regularly changing Web content. RSS feeds benefit users who want to subscribe to timely updates from favored Web sites or to aggregate feeds from many sites into one place. RSS can be seen as a publish-subscribe system for Web micronews. It suffers from the escalating bandwidth demand of RSS readers [39].

WS-Notification is a family of specifications that define a standard for topic-based notifications in Web services [21], [42]. It includes a standard message exchanges to be implemented by providers that wish to participate in notifications, standard message exchanges for a notification broker (allowing publication of messages from entities that are not themselves service providers), operational requirements expected of service providers and requestors that participate in notifications, and an XML model that describes topics. WS-Notification follows the Push-1:N-Explicit pattern (Fig. 3b). The notification protocols proposed in this paper follow the Push-1:N-Implicit pattern (Fig. 3d).

The Web Services Resource Framework (WSRF) is a family of OASIS-published specifications that allow Web services to become stateful [3]. Web services communicate with resource services (called WS-Resources) which allow data to be stored and retrieved. The framework relates to well-known Web services standards such as WS-Notification. From the perspective of WS-Notification, WSRF provides useful building blocks for representing notifications. From the perspective of WSRF, WS-Notification extends the utility of WS-Resources by allowing requestors to ask to be asynchronously notified of changes to resource property value [36].

WS-Eventing [9] is a major competitor of WS-Notification. Comparative studies of both specifications are presented in [22] and [35]. Both studies agree that WS-Eventing and WS-Notification are adopting ideas and concepts from each other and getting more mature with each update. There is currently a trend of convergence of both specifications. Proposals for creating a new standard that will integrate functions from WS-Notification with WS-Eventing are underway [14].

Our aim in this paper is to define a framework that includes several notification patterns for event-based SOAs. Existing standardization efforts such as WS-Notification and WS-Eventing implement one of the patterns described in the framework, namely, the Push-1:N-Explicit (both centralized and peer-to-peer). The patterns introduced in our framework do not contradict or compete with each other, but rather complement each other. We envision that a given SOA may implement several patterns (e.g., push and pull patterns) to cope with the requirements and peculiarities of its various applications and users.

The WS-Messenger project supports both WS-Eventing and WS-Notification specifications at the same time through a mediation approach [23]. The mediation techniques reconcile the differences between WS-Eventing and WS-Notification. WS-Messenger detects which specification the incoming SOAP messages use and processes them accordingly. Response messages follow the same specifications as request messages. Event producer can publish event notifications using either the WS-Eventing or the WS-Notification specification. This makes no difference to the event consumers since WS-Messenger performs mediations automatically.

Several techniques have adopted the concept of ontology in event-based systems. An ontology-based matching algorithm for content-based Push-1:N-Explicit dissemination is introduced in [44]. A Push-1:N-Explicit dissemination system which utilizes ontology to classify and query published data is presented in [40]. The approach described in [45] uses ontology to model events, topics, and subscriptions in content-based Push-1:N-Explicit dissemination systems. However, [40], [44], and [45] deal with the Push-1:N-Explicit pattern. In this paper, we use ontology for enabling the Push-1:N-Implicit pattern.

A publish-subscribe infrastructure for SOAs is proposed in [34]. The infrastructure is based on the Quicksilver platform developed by the same authors. The proposed infrastructure focuses on enabling IP multicast and providing reliability and QoS guarantees for the publish-subscribe model. Our work focuses on a different dissemination pattern, i.e., implicit. Some of the techniques (e.g., multicast) used in [34] can be adopted in our work. WebBIS defines an event-based protocol for Web services [30]. However, it focuses on propagating changes from component to composite services.

8 Conclusion

In this paper, we have dealt with dissemination patterns and protocols in event-based SOAs. Based on the interaction and filtering taxonomies, we propose a categorization of dissemination patterns in SOAs. One of the patterns we emphasize is implicit notification. This pattern uses the INO to model interaction interests among categories of services. We introduce three implicit notification protocols to deal with the issues of notification loops and repetition: centralized, distributed, and header-based. Finally, we describe a prototype implementation and conduct experiments to compare the proposed protocols.

As future research, we will develop techniques (e.g., machine learning [31] and mass collaboration [27]) to (semi)automatically define and maintain the INO ontology. We will also explore other protocols to minimize both notification time and the number of notifications. For instance, we will investigate the use of distributed hash tables [32]. Finally, we will define models and protocols for enabling the different dissemination patterns introduced in this paper.

Acknowledgments

This work was supported in part by grants from the University of Michigan's Office of the Vice-President for Research (OVPR) and the University of Michigan—Dearborn's Centre for Engineering Education and Practice (CEEP). The author would like to thank Shamil Hadi and Vasumathi Raman for implementing the prototype.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraj, *Web Services: Concepts, Architecture, and Applications.* Springer Verlag, June 2003.

[2] S. Baehni, R. Guerraoui, B. Koldehofe, and M. Monod, "Towards Fair Event Dissemination," *Proc. 27th Int'l Conf. Distributed Computing Systems Workshops (ICDCSW '07),* June 2007.

[3] T. Banks, *Web Services Resource Framework (WSRF)—Primer,* OASIS Committee Draft 01, Dec. 2005.

[4] A. Berfield, P.K. Chrisanthis, and A. Labrinidis, "Automated Service Integration for Crisis Management," *Proc. First SIGMOD Workshop Databases in Virtual Organizations (DIVO '04),* June 2004.

[5] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific Am.,* vol. 284, no. 5, pp. 34-43, May 2001.

[6] E. Bertino and K. Ramamritham, "Guest Editors' Introduction: Data Dissemination on the Web," *IEEE Internet Computing,* vol. 8, no. 3, May/June 2004.

[7] K. Birman and R. Cooper, "The ISIS Project: Real Experience with a Fault Tolerant Programming System," *Operating Systems Rev.,* vol. 25, no. 2, Apr. 1991.

[8] K. Birman, A.-M. Kermarrec, K. Ostrowski, M. Bertier, D. Dolev, and R. van Renesse, "Exploiting Gossip for Self-Management in Scalable Event Notification Systems," *Proc. 27th Int'l Conf. Distributed Computing Systems Workshops (ICDCSW '07),* June 2007.

[9] D. Box, L.F. Cabrera, C. Critchley, F. Curbera, D. Ferguson, S. Graham, D. Hull, G. Kakivaya, A. Lewis, B. Lovering, P. Niblett, D. Orchard, S. Samdarshi, J. Schlimmer, I. Sedukhin, J. Shewchuk, S. Weerawarana, and D. Wortendyke, "Web Services Eventing (WS-Eventing)," *W3C Member Submission,* Mar. 2006.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[11] S. Ceri, R. Cochrane, and J. Widom, "Practical Applications of Triggers and Constraints: Success and Lingering Issues (10-Year Award)," *Proc. 26th Int'l Conf. Very Large Data Bases (VLDB '00),* Sept. 2000.

[12] K.M. Chandy, B.E. Aydemir, E.M. Karpilovsky, and D.M. Zimmerman, "Event-Driven Architectures for Distributed Crisis Management," *Proc. Int'l Conf. Parallel and Distributed Computing and Systems (PDCS '03),* Nov. 2003.

[13] H. Chesbrough and J. Spohrer, "A Research Manifesto for Services Science," *Comm. ACM,* vol. 49, no. 7, July 2006.

[14] K. Cline, J. Cohen, D. Davis, D.F. Ferguson, H. Kreger, R. McCollum, B. Murray, I. Robinson, J. Schlimmer, J. Shewchuk, V. Tewari, and W. Vambenepe, *Toward Converging Web Service Standards for Resources, Events, and Management,* white paper, Mar. 2006.

[15] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys,* vol. 35, no. 2, pp. 114-131, June 2003.

[16] P.T. Eugster, P. Felber, and F. Le Fessant, "The 'Art' of Programming Gossip-Based Systems," *Operating Systems Rev.,* vol. 41, no. 5, Oct. 2007.

[17] D. Fensel, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce.* Springer Verlag, Sept. 2003.

[18] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure,* second ed. Morgan Kaufmann, Nov. 2004.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Oct. 1994.

[20] A. Geppert and D. Tombros, "Event-Based Distributed Workflow Execution with EVE," *Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware '98),* Sept. 1998.

[21] S. Graham, D. Hull, and B. Muray, *Web Services Base Notification 1.3 (WS-BaseNotification),* OASIS Standard, Oct. 2006.

[22] Y. Huang and D. Gannon, "A Comparative Study of Web Services-Based Event Notification Specifications," *Proc. Int'l Conf. Workshops Parallel Processing (ICPPW '06),* Aug. 2006.

[23] Y. Huang, A. Slominski, C. Herath, and D. Gannon, "WS-Messenger: A Web Services-Based Messaging System for Service-Oriented Grid Computing," *Proc. Sixth IEEE Int'l Symp. Cluster Computing and the Grid (CCGrid '06),* May 2006.

[24] V. Kolovski, Y. Katz, J. Hendler, D. Weitzner, and T. Berners-Lee, "Towards a Policy Aware Web," *Proc. Semantic Web and Policy Workshop (SWPS '05),* 2005.

[25] L. Liu, C. Pu, and C. Hsu, "Continual Queries for Internet Scale Event-Driven Information Delivery," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 4, July/Aug. 1999.

[26] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, "Bringing Semantics to Web Services: The OWL-S Approach," *Proc. First Int'l Workshop Semantic Web Services and Web Process Composition (SWSWPC '04),* July 2004.

[27] R. McCann, A. Doan, V. Varadaran, A. Kramnik, and C. Zhai, "Building Data Integration Systems: A Mass Collaboration Approach," *Proc. Int'l Workshop Web and Databases (WebDB '03),* June 2003.

[28] S.A. McIlraith, T.C. Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent Systems,* vol. 16, no. 2, Mar./Apr. 2001.

[29] B. Medjahed and D. Carver, "An Enterprise Messaging Middleware for Mobile Services," *Proc. 27th Int'l Conf. Distributed Computing Systems Workshops (ICDCSW '07),* June 2007.

[30] B. Medjahed, B. Benatallah, A. Bouguettaya, and A. Elmagarmid, "WebBIS: An Infrastructure for Agile Integration of Web Services," *Int'l J. Cooperative Information Systems,* vol. 13, no. 2, June 2004.

[31] T.M. Mitchell, "Machine Learning and Data Mining," *Comm. ACM,* vol. 42, no. 11, Nov. 1999.

[32] M. Naor and U. Wieder, "A Simple Fault Tolerant Distributed Hash Table," *Proc. Second Int'l Workshop Peer-To-Peer Systems (IPTPS '03),* Feb. 2003.

[33] National Research Council, *Improving Disaster Management: The Role of IT in Mitigation, Preparedness, Response, and Recovery.* Nat'l Academy Press, 2007.

[34] K. Ostrowski, K. Birman, and D. Dolev, "Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification," *Int'l J. Web Service Research,* vol. 4, no. 4, 2007.

[35] S. Pallickara and G. Fox, "An Analysis of Notification Related Specifications for Web/Grid Applications," *Proc. Int'l Conf. Information Technology: Coding and Computing (ITCC '05),* Apr. 2005.

[36] M.P. Papazoglou, *Web Services: Principles and Technology.* Prentice Hall, Sept. 2007.

[37] Y. Qi, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and Managing Web Services: Issues, Solutions, and Directions," *The VLDB J.,* vol. 17, no. 3, Mar. 2008.

[38] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel, "Web Service Modeling Ontology," *Applied Ontology,* vol. 1, no. 1, 2005.

[39] D. Sandler, A. Mislove, A. Post, and P. Druschel, "FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification," *Proc. Fourth Int'l Workshop Peer-To-Peer Systems (IPTPS '05),* Feb. 2005.

[40] J. Skovronski and K. Chiu, "Ontology Based Publish Subscribe Framework," *Proc. Eighth Int'l Conf. Information Integration and Web-Based Applications Services (iiWAS '06),* Dec. 2006.

[41] J. Spohrer and D. Riecken, "Services Science (Guest Editors Introduction)," *Comm. ACM,* vol. 49, no. 7, July 2006.

[42] S. Vinoski, "Web Services Notifications," *IEEE Internet Computing,* vol. 8, no. 3, May/June 2004.

[43] X. Yang, A. Bouguettaya, B. Medjahed, H. Long, and W. He, "Organizing and Accessing Web Services on Air," *IEEE Trans. Systems, Man, and Cybernetics—Part A,* vol. 33, no. 6, Nov. 2003.

[44] J. Wang, B. Jin, and J. Li, "An Ontology-Based Publish/Subscribe System," *Proc. Fifth ACM/IFIP/USENIX Int'l Middleware Conf. (Middleware '04),* Oct. 2004.

[45] J. Wang, B. Jin, J. Li, and D. Shao, "A Semantic-Aware Publish/Subscribe System with RDF Patterns," *Proc. 28th Ann. Int'l Computer Software and Applications Conf. (COMPSAC '04),* Sept. 2004.

Brahim Medjahed received the PhD degree in computer science from Virginia Tech in May 2004. He is an assistant professor in the Department of Computer and Information Science, University of Michigan—Dearborn. His research interests include service-oriented architectures, distributed computing, semantic Web, and databases. He received the 2004 "Outstanding Graduate Research Award" at Virginia Tech's Department of Computer Science. He guest edited a special issue of the *ACM Transactions on Internet Technology* on semantic Web services. He has served on numerous conference program committees. He is the author of more than 40 publications. He is a member of the IEEE, the IEEE Computer Society, and the ACM.