AwareDAV: A Generic WebDAV Notification Framework and Implementation

Henning Qin Jehøj
Dept. of Computer Science
University of Aarhus
Denmark
jehoej@daimi.au.dk

Niels Olof Bouvin
Dept. of Computer Science
University of Aarhus
Denmark
n.o.bouvin@daimi.au.dk

Kaj Grønbæk
Dept. of Computer Science
University of Aarhus
Denmark
kgronbak@daimi.au.dk

ABSTRACT

WebDAV needs awareness support in order to be a full-fledged collaboration system, This paper introduces AwareDAV, a new WebDAV extension framework enabling shared awareness through event notification. By extending the WebDAV protocol with seven new request-methods and an extensible XML based event subscription scheme, AwareDAV supports fine grained event subscriptions over a range of transport mechanisms and enables a wide range of collaboration scenarios. This paper describes the design of AwareDAV, its API, experiences with its initial implementation, as well as a comparison with Microsoft Exchange and WebDAV-notify.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Applications; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work

General Terms

Standardization, Human Factors

Keywords

WebDAV, Event Notification, CSCW, AwareDAV

1. INTRODUCTION

As the largest document repository ever, the Web is becoming the environment where work is done. With this shift, the need for proper support of collaborative work has become pressing, leading to standards such as the HTTP extension WebDAV[10]. WebDAV enables authors to upload, lock, annotate and version resources residing on Web servers, thus facilitating remote collaboration.

However, while a marked improvement over the existing HTTP protocol, WebDAV is still lacking in a number of areas crucial for CSCW applications. Most important of these is shared awareness through event notification. Collaborators must in order to smoothly coordinate their work continually be aware of each others' activities. This cannot be handled satisfactorily with the basic WebDAV constructs. Locking is an essential mechanism to ensure that collaborators do not accidently overwrite each others' changes, but as implemented in WebDAV it still relies on an out-of-channel mechanism (or simple polling) to alert co-workers that a resource has become available again. This goes for the rest of WebDAV—the essential functionality for collaboration is present, but all require

participants to actively monitor a site or resource in order to discover changes as they occur. A more elegant approach would be to have the WebDAV server alert the users to relevant changes as they occurred. This is known as a notification service, and this paper focuses on the lack of a proper notification mechanism and proposes AwareDAV, an extension to WebDAV, providing an extensible set of awareness services.

The paper is structured as follows: Section 2 describes related work in the field of CSCW; Section 3 describes WebDAV and the work done within the WebDAV community; Section 4 details the proposed WebDAV extensions to facilitate notifications; Section 5 describes the detailed design at message and XML level; Section 6 provides two API level examples of use of AwareDAV; our experiences with the implementation of the AwareDAV framework is described in Section 7; Section 8 compares AwareDAV with two existing WebDAV notification frameworks, and the paper is concluded in Section 9.

2. CSCW AND NOTIFICATION SERVICES

Work is rarely done exclusively by the individual; there is usually a social context within an organization that determines why work is done, what work is done, and how it is carried out. This social context necessitates cooperation and coordination between workers, be it on the same level of the organization or between levels. The purpose of computer supported cooperative work (CSCW) is to investigate and understand this cooperation, and to support it with computers.

Cooperation requires coordination, and coordination requires a level of shared awareness of what co-workers are doing. As described by Heath and Luff [11] in their analysis of work done by traffic controllers in the London Underground, maintaining shared awareness between co-workers can be crucial. This need lends itself naturally to notification services. Many systems have endeavored to support this: The GroupDesk system [9], the NESSIE system [14], the Elvin system [7], the AREA system [8], and the BABBLE system [3] to mention a few. These systems seek to support shared awareness through means such as messaging, chat, and event notification. Some aspects of awareness support require explicit actions by the user, such as sending a message to another user, while others are implicit. The latter form is for instance the action of updating a file in a CVS repository, which results in a broadcast to all users of an Elvin system, or joining a topic in BABBLE, which shows other users that you are active. Some systems, notably AREA and Elvin, attempt to integrate with other applications, so that actions in these applications are broadcast to other users. In recent years, event notification systems have been extended to explore issues related to events across wide-area networks [4].

3. THE WEB AND WEBDAV

The Web was designed with a client/server architecture with transient connections and no client state residing on servers. This made it well suited for large scale distribution of documents. Transparent connections were later introduced to speed up transfer of complex Web pages. Even though the first Web browser was also an editor, most popular browsers were and still are just viewers, thereby avoiding the complexities of collaborative remote editing of Web pages. The only support for remote authoring in HTTP was the PUT method facilitating upload of a document. However, without any knowledge of the actions of other clients, such updates could easily cause a recent update (from another client) to be overwritten. Thus, HTTP had two major weaknesses: Document control and awareness support.

3.1 WebDAV Standards

WebDAV [10] tried to address the authoring problems of the Web, especially the document control problem. It ensured that documents were arranged into hierarchies, which could be manipulated through moving and copying resources from one location to another. It introduced locks to prevent accidental lost updates. Finally, it introduced properties, which could contain arbitrary metadata about the resources, e.g., owner or date of creation. This provided some awareness support, as clients could now gain insight into the age and ownership of documents.

The Versioning Extensions to WebDAV (ΔV[5]) enriched document control to version control, also a well known method to avoid data loss. Due to its complexity, this extension was divided into several features, allowing servers to choose a sensible and functional subset. This implies that a client has to query the server about its abilities in order to use its version extensions. This division was necessary to accommodate the different versioning scenarios while still maintaining a common protocol.

Privacy issues on the Web have hitherto been handled through authentication and encryption, but the introduction of distributed authoring raises the need for assigning different access rights for different operations, e.g., read or write. The Access Control Protocol[6] defines both a set of access rights and the properties to examine and change them. As servers may have different access rights depending on the underlying platform, the access right scheme has been somewhat loosely defined, and an inclusion ordering between access rights introduced (e.g., if a client has been granted write access, that might include property modify access as well). A client can then after querying the scheme choose the access rights it is familiar with.

Until the introduction of WebDAV Search[15], the only way of locating a certain resource, based on its property values, was to retrieve the properties from the resource hierarchy and search locally on the client. The WebDAV Search protocol introduces the SEARCH request enabling server-side searching. It provides a basic search grammar, which can be used for searching, but a server could also support other grammars as the mechanism is extensible.

3.2 WebDAV Implementations

WebDAV has been implemented for several Web server platforms, and used in many clients in more or less compatible ways. Some of these are Apache/mod_dav, Oracle Internet File System, Jigsaw, Xythos Storage Server, Microsoft IIS 5/6 and Microsoft Exchange.

Microsoft Exchange's WebDAV [13] implementation already includes notification support through both pushing and polling of notifications. Polling is done through its HTTP extension and a push mechanism is implemented through UDP datagrams. As datagrams are inherently unreliable, an acknowledgment through polling is required. Notifications state only that something has occurred rather than providing all the details (as opposed to WebDAV-notify). This requires only small amounts of information to be sent, however, if full knowledge is required at least 3 messages has to be sent to the client: the notification, the acknowledge, and finally a GET or/and PROPFIND request, and even then the client has only the latest state of the resource, not *what* changed, The requests SUBSCRIBE and UNSUBSCRIBE are used to manage subscriptions. Notification is polled with the POLL request. Events are matched with subscriptions based on type and resource URI with depth. Supported types are **update**, **update/newmember**, **delete**, **move**, and **newmail**.

WebDAV-notify[12] is a work in progress from the Jabber Software Foundation, which seeks to create an event notification system for WebDAV. This is done by extending the XMPP-PubSub protocol (an extension of XMPP-CORE), which is a XML publish/subscribe protocol. The primarily focus is to create a near real time and secure notification mechanism. This is accomplished by:

- Mapping the WebDAV service to the notification service, collections to collection-nodes, and resources to leaf-nodes, i.e., WebDAV's resource model to XMPP-PubSub's node model.

- Making a persistent connection (currently only through TCP sockets) from the client to the notification service, which is used both for subscriptions and notifications.

- Reflecting the corresponding nodes in resources, allows clients to subscribe for resource-change notifications and node-creation notifications at nodes.

WebDAV-notify supports both keeping end users aware of changes as well as enabling co-servers (e.g., caches or mirrors) to keep track of changes. In order to do this without requiring the co-servers to request extra information, a set of event payloads for basic WebDAV request are defined, including a diff element, which makes it possible to keep mirrored resources in sync.

Microsoft Exchange and WebDAV-notify have been designed to suit their specific needs. Neither, however, are fully general and each make demands on implementors that may be unreasonable. See Section 8 for a more detailed discussion on these two systems and a comparison with AwareDAV.

3.3 WebDAV Shortcomings

A Web client can, due to the disconnected nature of the Web, exert only a limited amount of control over a server's resources. Locking and versioning in WebDAV have yielded some control to the client, but locks are still coarse grained, and versioning cannot always avoid unresolvable conflicts.

Awareness is supported through explicit polling. The Exchange extension and the WebDAV-notify draft support different notifications mechanisms. Exchange uses unconnected communication, and does not provide much information in the notification. WebDAV-notify requires a fixed connection or gateways and uses two different and not quite compatible resource models. They are both somewhat inflexible and not for general use.

4. THE AWAREDAV FRAMEWORK

Having established the limitations of WebDAV standards and implementations, we present AwareDAV, a generic extension of WebDAV for awareness support. This is accomplished by three awareness features, each extending the area of awareness.

(a) Event awareness   (b) Subject awareness   (c) Intersubjectivity
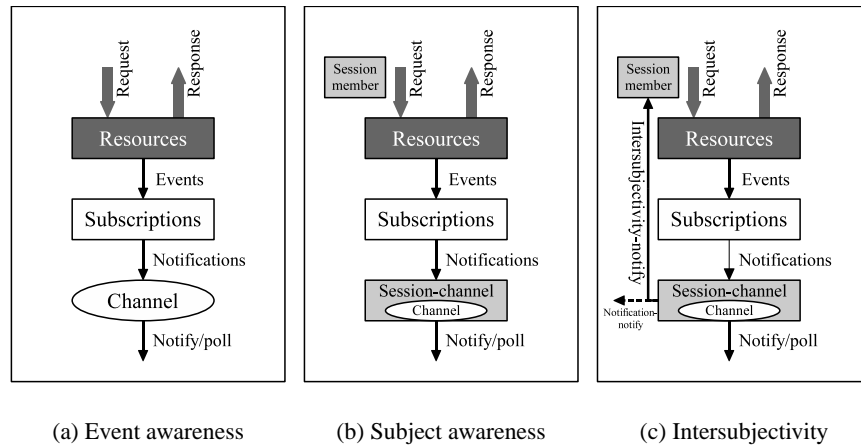
Figure 1: AwareDAV event flow models

4.1 Awareness Features

This section describes the level of awareness supported by the AwareDAV framework.

4.1.1 Event Awareness

Event awareness, as seen in Figure 1(a), enables a server to notify clients about events occurring in the data model (i.e., the resource hierarchy). Whenever a successful WebDAV request is issued, the server performs the corresponding actions in a more or less atomic way with each action generating an event. If a client is interested in certain resources, it must subscribe for events at these resources. The subscription will catch desired events and notify the client accordingly. The subscription determines whether to keep the notification in its notification queue waiting for the client to look at the queue or to dispatch it immediately to the channel. A channel could be a e-mail address, a TCP socket, or something else.

4.1.2 Subject Awareness

A crucial element of shared awareness is knowing *who* is doing what. Subscriptions as formulated in AwareDAV have an owner and a channel, but these may just be URIs, and does not necessarily divulge who is doing what.

Subject awareness, as shown in Figure 1(b), introduces sessions and session-channels, which provides a framework for structuring work into sessions, and to announce current work and interest in a certain session by *logging-in* to it. Session-channels come in three flavors: independent, session-members, and attached. A session-member represents an active user (or actor) similar to ACL [6]. Therefore each client should use the **session-channel** header to identify the acting session-channel.

But as only active members can receive notifications, a system supporting off-line notifications should use session-channels attached to the session-member to represent a single session-member.

A session-channel can represent channels, as all session channels have a default channel, as shown in Figure 1(b). The channel (its queue, and its lifetime) is then be shared between subscriptions using the same session-channel.

A session channel is a "special" resource which should be inside a session (a collection type resource). As sessions are hierarchical organized, all sessions will be related to its parent (either the session-root or another session). Each session may have a data-collection, which may be controlled by the session limiting access from non-session-channels.

4.1.3 Intersubjectivity Awareness

With the mechanisms described above, we can be kept aware of what is going on and who is doing what, but it would be even better if we also knew what others knew. This feature enables a client to "hear" what others are being told about it.

Intersubjectivity as described in the iScent framework [1] is the feature that *you could be notified when others were notified about your actions*. This is desirable in a tight cooperative setting, because if you know what others already have been notified about, you can work with them based on that knowledge.

This can be generalized to catching "notified" events on a session-channel as shown in Figure 1(c). These notifications work by having a spying subscription on a session-channel or by setting the **Mutual-Notify** header requesting a notified notification from all generated notifications. The last option is not only the most useful but also the way to catch notifications for "anonymous" channels.

4.1.4 Computation Extension

While the three awareness mechanisms described above will cover many use scenarios, AwareDAV also has a mechanism for extensions, which extends the subscription with computational power, so that it can act upon successful event matches. The subscription could e.g., grab a lock to a highly requested resource when it is unlocked. The event matching mechanism *should* therefore ensure fairness in the order it process subscriptions: The subscriptions eldest and closest to the event should be matched first.

The engine is also used in the **where**-expression, granting detailed matching capabilities. It is very similar to the SEARCH request from DASL [15].

4.2 Awareness Architecture

Our design should be architecturally independent. The protocol should be usable both in a small office running Microsoft Exchange, and on a large multi server configuration. We must therefore consider channels and server setup:

Channels: The channel is a communication conduit to the client given by a XML representation. This could be email, a TCP socket, or something else. Internally, the server setup can

use any protocol. The possible channel types of a notification server *should* of course be obtainable. Connected communication like sockets will probably only be used in tight and relatively small groups. Email could be used with low-rate notification scenarios where the user want to be notified even when off-line. A third alternative would be to use an IM service, such as Jabber.

Servers: In a reasonably sized setting, it would be natural to off-load the notification job from the document server to a dedicated notification server. In a large scale scenario, we might even need more notification servers, as well as many document servers. The document server would then route the relevant requests to the notification server. This should work well, as long as a client does not need to inquire for existing subscriptions, or to refresh it subscriptions very often.

In these cases we could reuse a request redirection. Namely the response 300 MULTIPLE CHOICES, which uses the *Location* headers to give a list of URL's refering to the notification server. It would additionally use the *Allow* header to specify the set of methods which should be redirected. A simple client, having only limited traffic to the notification server, could of course just send everything to the primary Web server, and be redirected every time.

Requests can be redirected if they address a subscription (by URI and subscription ID) or/and a session-channel (identified by the URI), which are the two entities it makes sense to offload.

4.3   Event Processing

Both events and subscriptions are tied to the resource hierarchy. Subscriptions are sub-objects of resources in the same way as locks are, and are inherited to a specified depth just like locks. Events emerge at those resources, which are affected by the corresponding actions. These resources are called the event origin. So event processing begins with matching, and if successful, continues like this:

1. Event subscription matching:

    (a) The event-origin *must* match the coverage of the subscription (i.e., have a resource in common).

    (b) The event types of the event *must* match the subscription (i.e., the subscription *must* listen for at least one of the types of the event).

    (c) The event *must* match the **where**-expression of the subscription. (The **where**-expression *must* evaluate to true. It works like the **search**-expression of DASL[15])

2. Notification construction:

    (a) Event-property filtering: The subscription will select the needed event-properties and insert them into the notification. (This to avoid huge notifications).

    (b) The actions expression (if any) is evaluated/run and the result added to the notification.

    (c) The properties, which are requested from the subscription, are fetched from the resource and added to the notification. (Note: Even if a event-property is mirrored in a resource-property the action could change it).

3. The notification will be put onto the channel, which will dispatch it according to its nature. Some channels has queues which holds the notification until it is polled or packed in a delayed notification-set.

Most event matching can be accomplished with the two first match mechanism.

4.3.1   Event Type Scheme

The event types scheme is analogous to access rights from ACL[6] and designed to be extensible. Due to the extensibility, events will often have more types. An simple client may not recognize new event types on an special event but can still recognize the basic types within it and therefore work with more advanced servers.

The type scheme consists of the following: **created-x**, **deleted-x**, **updated-x**, **modified-x**, **refreshed-x**, **read-x**, **moved-x**, and **failed**.

Objects that **x** may span over are: **Resources**, **contents**, **properties**, **bindings**, **locks**, **subscriptions**, **notifications**, and **session-channels**. This can of course be extended by other protocols, maybe with **sessions**, **versions**, **version-histories**, etc. The **failed** event is not differentiated based on its object. The **refreshed-x** event types may be omitted if the server guarantees that the **delete-x** event is sent when the objects timeout. The actual names used can be seen in Table 1.

Extensions are required to extend the event system in a consistent way in regard to the scheme described here.

An example could be VERSION-CONTROL: This request (from ΔV [5]) puts a resource under version control. This really ought to be invisible for version agnostic clients, so it should not use any of the basic event types.

The CHECKOUT of a version creates a new (version controlled) resource and should therefore emit a **created** event.

4.3.2   Event Properties

Every event carries some extra information in its properties. This is an extensible mechanism just like resource properties, and each property is an XML element. Basic properties include the method and resource type. This is why we do not have event types like **created-collection** and **created-session**, and why there is only one failed event. See Section 5.5 for other event properties.

A subscription can match against these properties if the **where** expression is supported. And to avoid transmitting more than necessary, a subscription may select the interesting properties using its filtering mechanism.

5.   CORE PROTOCOL DESIGN

In this section we list the AwareDAV protocol method- and XML-syntax, and discuss its interoperability with DASL[15] and ACL[6]. Throughout all XML examples and definition we use the prefix `d:` for the namespace `DAV:` and the prefix `a:` for the namespace `http://www.daimi.au.dk/AwareDAV`.

5.1   Methods

We have extended the WebDAV protocol with seven new request methods: SUBSCRIBE, UNSUBSCRIBE, POLL, NOTIFY, LOGIN, LOGOUT, and SESSION-CONTROL. The first four are inspired by Microsoft Exchange[13], and the rest enable subject awareness.

SUBSCRIBE

A SUBSCRIBE request is used to subscribe to events occurring at the resource or in the collection hierarchy (the coverage of the subscription) specified by the request URL and the optional *Depth* header, or to refresh existing subscriptions.

- A refresh SUBSCRIBE request has no body and *must* have at least one *Subscription-ID* header.

- A create SUBSCRIBE request has no *Subscription-ID* headers and the body *must* be a **a:subscribeinfo** XML element. This element specifies who owns the subscription, which events it will listen to, and how it will respond to these events. The

Action	Resources	Contents	Properties	Bindings	Locks	Subscription	Notification	Active server-channel
create	created	-	-	bound	locked	subscribed	notified[a]	logged-in
delete	deleted	-	-	unbound	unlocked	unsubscribed	-	logged-out
update	updated	updated-content	-	-	-	-	-	-
modify	-	-	modified-properties	-	-	-	-	-
read/copy	copied	read-content	read-properties	-	-	-	polled	-
move	moved	-	-	-	-	-	-	-

[a]Notified is a special event type generated by the NOTIFY method or intersubjectivity. Notified-type notifications do not generate additional notified-type events.

Table 1: Basic event types

request *may* fail if the request-URL is an unmapped request-URL. The *Subscription-ID* is returned in the header.

SUBSCRIBE, UNSUBSCRIBE, and POLL requests *may* fail, if the request-URL is not within the coverage of the subscription(s). The *Timeout* header may be used to urge a certain lifetime upon the channel, and should be used to pass the resulting lifetime in the response. The **a:subscribeinfo** XML element is defined in Section 5.2.

UNSUBSCRIBE

An UNSUBSCRIBE request is used to remove subscriptions made with SUBSCRIBE requests.

- An UNSUBSCRIBE request has no body and *must* have at least one *Subscription-ID* header.

POLL

The POLL request is used to poll the notification queue of a subscription in the cases where server-to-client communication is unavailable or unreliable.

- A POLL request has no body and *must* have at least one *Subscription-ID* header.

- The successful POLL response *must* have a **a:notification-set** body.

The **a:notification-set** XML element is defined in Section 5.3.

NOTIFY

The NOTIFY method simply generates a user defined notification, which is transmitted to the specified recipient. This is similar to email or IM functionality, but has the advantage of being observable by others.

- A NOTIFY request *must* have a **a:notify** body. The request-URL *must* specify a *session-channel*, which receives the notification generated from the body.

The **a:notified**-type event *must* be generated at the session-channel upon successful completion. The **a:notify** XML element is defined in Section 5.3.

LOGIN

The LOGIN method is used to login (and activate) a *session-channel* resource in a *session*. It is also used to refresh the login.

- A refreshing LOGIN request has no body and the request-URL refers to an existing active session-channel. This channel and its subscriptions will be refreshed according to the *Timeout* header.

- An activating (and creating) LOGIN request has a **a:login** body and the request-URL refers to either an inactive session-channel or a session.

- A activating LOGIN response has a **a:login-response** body with an list of re-subscribed subscriptions if any. The *Location* response-header indicates whether a session-channel was created.

```
<!element a:login (a:channel, d:displayname?, a:session-
        member?, a:channel-type?, a:mutual-notify?)>
<!element a:session-member (d:href)?>
<!element a:channel-type (a:online-channel,
        a:offline-channel)>
<!element a:login-response (a:subscription-list?)>
```

The optional **a:session-member** element specifies the type of the created session-channel. If **a:session-member** is not present it is independent; if it is empty it is a session-member; and if it contains a **d:href** it is attached to the session-member given by the that URL.

The lifetime of active session-channels is inherited by all subscriptions using them. If the specified communication channel is another session-channel a copy is made including associations to attached session-channels. Intersubjectivity may be forced upon every request made by the active session-channel (**a:mutual-notify**).

LOGOUT

The LOGOUT method is used to logout (and deactivate) a *session-channel* resource.

- A LOGOUT request has a **a:logout** body.

- A LOGOUT response has a **a:logout-response** body with an list of auto-unsubscribed (and auto-subscribe capable) subscriptions if any.

```
<!element a:logout ()>
<!element a:logout-response (a:subscription-list?)>
```

All subscriptions owned by the session-channel *must* be automatically unsubscribed upon successful completion.

SESSION-CONTROL

The SESSION-CONTROL method is used to create sessions or change the control options of an existing session. If the request-URL is unmapped a new session should be created, and if it represents an existing session it should alter the options of the session. A session must logically reside in another session or in a session-root.

- A SESSION-CONTROL request has a **a:session-control** body specifying the session-parent, the data collection, and control options.

- A SESSION-CONTROL response has a **a:session-control-response** body and uses a *Location* header for a newly created session.

```
<!element a:session-control (a:session?, d:displayname?,
        a:data-url?,a:data-control?,a:mutual-notify?)>
<!element a:session (d:href)>
<!element a:data-url (d:href)>
<!element a:data-control (a:auto-login?,a:subscribe?,
        d:read?,d:write?)>
<!element a:session-control-response ()>
```

Implementations may allow a session to control the data collection by requiring users to be logged-in in order to read, write, or subscribe to resources in the data collection. They may even support auto-login when it is required. Intersubjectivity may also be forced upon session-channels within the session.

These seven request-methods form the AwareDAV API, together with the XML-structures defined in the following sections.

5.2 Subscriptions

A subscription is a sub-object of a resource, and is identified by the resource-URL and a subscription-identifier. Even so the identifier of a subscription *should* be unique. It may have both a user supplied owner (i.e., **d:owner**), a formal owner (i.e., a session-channel), and optionally an attached session-channel.

The main content of subscriptions is specified by the **a:subscribe-info** XML element below: The events it is listening for is determined by event-types (**a:what**) and the optional match-expression (**a:where**); the content of notifications is specified by event-property names (**d:select**) and resource-property names (**d:propfind**); the side-effect of notifying is specified by a small program (**a:action**); and how notification should be dispatched is determined by the channel (**a:channel**). Supported channels may include TCP (**a:tcp**), e-mail (**a:email**), HTTP connection polling (**a:polling**), and session-channels (specified with **d:href**).

```
<!element a:subscribeinfo (d:owner?, a:what, a:where?,
        d:select?, d:propfind?, a:action?, a:channel)>
<!element a:what (ANY)> ; any event-type
<!element a:where (ANY)> ; any match-grammar
<!element d:select (d:allprop | d:prop)>
<!element a:action (ANY)> ; any action-grammar
<!element a:channel (ANY)> ; any channel-description
<!element a:polling (EMPTY)>
<!element a:email (d:href, a:delayed?, a:transform?)>
<!element a:tcp (d:href, a:port?)>
<!element a:subscription (d:href?, a:subscription-id?,
        d:owned-by?, d:owner?, a:what?, a:channel?)>
```

The following resource-properties exist in order to help use subscriptions:

- The **a:eventtype-discovery** property contains the supported event-types for this resource. Any SUBSCRIBE request with unknown event-types *must* fail; unsupported but known event-types *should* be silently ignored.

- The **a:channel-discovery** property contains a list of supported channels. E.g., TCP, email, or something else. A session-channel is supported if the default channel of it is supported.

- The optional **a:subscription-discovery** property contains a list a **a:subscription** elements, representing the subscriptions at the resource and their owners. It *may* collapse more subscriptions into one **a:subscription** not discarding any **a:owned-by** value.

As the XML definitions show, AwareDAV is extensible both in regard to event-types, matching, server-side programming, and communication protocols.

5.3 Notifications

When notifications are received, more notifications may have been packed together in a **a:notification-set** either because we are polling, or because the channel-specification allows for notification-delays. Each notification contains a subscription part, an event part and a notification computing part.

The subscription part contains identity and owner of the subscription. The event part (**a:event**) contains event-types (**a:what**) and event-properties (**d:prop**—see also Section 5.5). The computing part may consist of the result of the computed side-effect (**a:action-result** and retrieved properties from the resource of the subscription (**d:prop**).

```
<!element a:notification-set (a:notification*)>
<!element a:notification (d:href?, a:subscription-id?,
        d:owner?, a:owned-by?, a:event, a:action-
        result?, d:prop?)>
<!element a:event (a:what, d:prop)>
<!element a:action-result (ANY)>
<!element a:notify (d:prop)>
<!element a:message (ANY)>
```

The client may ignore unknown event-types in a notification and will not receive any unknown event-properties that it did not explicitly ask for.

Notifications may be due to an explicit subscription, or inter-subjectivity using the *Mutual-Notify* header, or an intersubjectivity-enabled session-channel. User initiated notifications (or messages), which are sent with the NOTIFY request, generate a **a:notified** event with the provided event-properties. The event-property **a:message** *should* be used as a general message for clients not understanding specialized properties.

5.4 Event Type Scheme

The event-type scheme (**a:what**) is already described in Table 1, but here we will specify the method-to-event relations and discuss the design choices. The events can be seen in Table 2 and Table 3.

The multiple type scheme was introduced as AwareDAV should be easy to extend with new event types, while maintaining compatibility with older clients. It also makes the choice between focusing on bindings or resources easy, as we choose both. ACL[6] had to make that choice as it would be troublesome to have different rights for creating and binding the chosen bindings (i.e., if you can move a resource somewhere, you should presumably be able to create a resource there as well).

The **modified-x** types are an union of **created-x**, **deleted-x**, and **updated-x**, which should be used when deletion is similar to assigning an empty value. The **modified-properties** is therefore the only event being generated from a PROPPATCH request.

Though refreshing (locks, subscriptions, session-channels, etc.) often updates some resource-properties, the **a:modified-properties** *should* not be used, as no significant changes occur. Instead the optional **refreshed-x** event-types may be used. If the server guarantees that **unlocked**, **unsubscribed**, or **logged-out** events are generated upon timeout, **refreshed-x** events may be omitted entirely.

Having listed the required event-types that *should* be supported, we now proceed with the properties of events.

5.5 Event Properties

Event-properties are unrelated with resource-properties (though they use the same **d:prop** XML element). Properties of events explain who caused it, when it was done, and what happened. Noti-

Method		Event type
GET		read-content
HEAD		(read-content)
PUT	(create?)	created, bound
	(update?)	updated-content
POST		?
DELETE		deleted, unbound
OPTIONS		-
PROPFIND		read-properties
PROPPATCH		modified-properties
MKCOL		created, bound
COPY	(source:)	copied
	(destination-create?)	created, bound
	(destination-update?)	updated
MOVE	(source:)	moved, unbound
	(delete-destination?)	deleted, unbound
	(destination)	moved, bound
LOCK	(create?)	locked
	(refresh?)	(refreshed-lock)
UNLOCK		unlocked
REPORT		(read-properties)

Table 2: Mandatory Events for WebDAV Methods

Method		Required events
SUBSCRIBE	(create?)	subscribed
	(refresh?)	(refreshed-subscription)
UNSUBSCRIBE		unsubscribed
POLL		(polled)
NOTIFY		notified
LOGIN	(create?)	logged-in, created, bound
	(refresh?)	(refreshed-channel)
	(update?)	logged-in
LOGOUT		logged-out, (deleted, unbound)

Table 3: Mandatory Events for AwareDAV Methods

fications will receive the actual properties filtered by the **d:select** property names.

The supported event-properties will vary depending on the purpose of the notification mechanism. Some clients need to know everything that changed, others may need to know some basic event-properties even if they did not change.

Standard event-properties are:

```
<!element d:owner (href?)>
<!element a:owned-by (href)>
<!element a:date (#PCDATA)>
<!element a:method (#PCDATA)>
<!element a:origin (d:href, d:etag?, d:depth?,
        d:multistatus?)>
<!element d:resourcetype (#PCDATA)>
<!element  a:unknown (#PCDATA)>
<!element a:src-origin (a:origin)>
<!element a:dest-origin (a:origin)>
<!element d:response (...)> ; from WebDAV
<!element d:activelock (...)> ; from WebDAV
<!element a:subscription-id ()>
<!element a:what (ANY)>
<!element a:message (ANY)>
<!element a:report (ANY)>
```

While the **d:owner** of locks and subscriptions is an arbitrary client-supplied element, the **d:owner** of events and resources is an ACL[6] principal. The **a:origin** of an event is specified by the root of its (**d:href**) and its **d:depth**; if the event failed on a subpart of the resource-hierarchy, the failed sub-branches are named in the **d:multistatus**. For a singleton origin, the **d:etag** may be present as well. The **d:resourcetype** may be **a:unknown** if it cannot be derived without extra resource-property retrieval. The **d:response** property indicates affected resource-properties, and the **a:report** property about used reports.

A server may support the **propertyupdate** property for PROP-PATCH and **diff** property for PUT, as WebDAV-notify[12] does.

In many cases, some event-properties do not make sense and will be omitted, e.g., **d:resourcetype** only makes sense on singleton origins. The **d:owner** property will be derived from authentication, and **a:owned-by** property will be determined by the used session-channel (i.e., by the *session-channel* header).

If a **d:select** of a SUBSCRIBE request lists unknown event-properties it *must* fail, with a DAV-error describing which event-properties are unsupported.

We have here described the extensible event-property mechanism. Similar to resource-properties the **d:allprop** filter will not return all event-properties, and it *should* only return the standard event-properties listed above.

5.6 Computations

Computations are used two places in AwareDAV, in the **a:where**-expression and in the **a:action**. These two expression vary much in their required vocabulary. The first *must* evaluate to a truth-value based upon the event-properties; the other perform some side-effect.

The **a:where**-expression may contain a **d:where** element from the basic search grammar of[15], where **d:prop** is interpreted as event-properties, or a server may implement other match-grammars.

The **a:action** may contain a **a:simple-action** element or another root element of an supported server-side languages. The **a:simple-action** language consists of a list of statements described below. Server may support other server-side languages as well, which may be discovered using the **a:supported-...** properties similarly to WebDAV Search [15].

```
<!element a:simple-action (ANY)> ; any statement below
<!element a:unsubscribe (EMPTY)>
<!element a:lock (EMPTY)>
<!element a:if (ANY)>
<!element a:supported-match-grammar-set (d:grammar*)>
<!element a:supported-comp-grammar-set (d:grammar*)>
```

This very simple language allows a subscription to unsubscribe itself and lock the associated resource. The language could easily be extended when other uses beyond "<if> <lock/> <unsub-scribe/> </if>" are discovered.

Even though these two simple approaches handle the job, a more ideal approach may be to create a common language for **a:where**-expressions, **a:action**'s, and searches (from DASL[15]).

5.7 Access Rights and Principals

The ACL protocol[6] introduced principals as representatives for actors and assigns to these actors access rights or specific resources. If an implementation supports both AwareDAV and ACL, we have to contemplate two issues: The relation between principals and session-members, and how access-rights affect notifications.

A session-channel is defined as a resource with a **a:session-channel** and an **ad:active** property, so a principal may well be a session-channel. But often a principal wants to use more session-channels. In these cases, a server *may* use a new resource type **a:session-channel**. These resource should be associated with a principal through the **d:owner** and **d:principal-url** property as defined in ACL [6].

This is so because users may want to join more sessions (creating session-channels) without having the possibility of adding principals. Sessions could then be created using the SESSION-CONTROL method, and session-channels with the LOGIN method. Another option for implementors is to prohibit session creation, only allowing the sessions (or collections) containing the principals, and in that case, principals and session-channels could be the same.

In general, if an actor has read access, he should also have subscribe access. But the coverage of a subscription may extend to resource and sub-objects that the actor has not access to, and he *must* be prevented from access to anything beyond his access level.

Thus, events will have to carry the access rights information of their origin, and match it with the access rights of the principals of the subscriptions. In case of multiple origins, a server could choose to restrict event-properties to inform about the origin root.

Some requests like SEARCH[15] may contain indirect information, which reveals too much (e.g., searching for "secret deals with Ted", which states that the actor and "Ted" has something going on). The events *should* therefore not contain such information.

As can be seen, combining ACL and AwareDAV raises some potential problems that must be addressed to make access rights control safe and efficient.

6. API USAGE EXAMPLES

We describe two examples to illustrate the utility of AwareDAV. One simple with a browser client using event awareness, and a second more complex using all three features.

Simple client

The simple client can connect to an AwareDAV server, browse the resource hierarchy, lock resources and view/edit documents (see Figure 2(a)). The interactions involved are

- Connecting involves opening a TCP socket, and sending the OPTION request to verify that the server is indeed an AwareDAV server. The DAV response header of the OPTION request will contain the keywords: `events`, `sessions`, and `intersubjectivity` depending on the level of AwareDAV support.

- Browsing the resource hierarchy involves opening and closing folders. For each open folder, a PROPFIND request with depth 1 is issued to retrieve the children, and a SUBSCRIBE request (also with depth 1) ensures that the client is informed of any changes to the displayed resource tree. The subscription will listen for created, deleted, copied, moved, bound, unbound, locked, and unlocked events. When the folder is closed, it should be unsubscribed.

- The client will then receive notifications in its model about changes and locks, and can reflect this in an explorer view.

- When viewing a resource, it is made read only (in the client) if it is locked. Otherwise, the user can be asked whether the resource should be locked for editing.

Advanced client

The advanced client extends the simple one with sessions and intersubjectivity. The user may choose a session to display and login to. Each time a intersubjectivity notification is received, the client can visualize it with a "sending message" animation on the session-member that was notified. The animation may differ depending on the kind of event-type that was caught by the session-channel. Thus, the user will immediately see who received notification of the user's actions, e.g., if they had the browser open at the same resource or maybe another type of client (see Figure 2(b)). The interactions involved are:

- The **session-tree** report is issued to find sessions, and the **session-channel-list** report is used to find session-members in them.

- Logging in or out should be reflected in the session view, so there will be a subscription for every open session.

- All requests should use the *Mutual-Notify* header in order to receive the intersubjectivity notifications. This can be done when the client has logged into the session-channel. These notifications will all be received by the session-channels' default channel.

- All other operations occur as before, perhaps with the slight modification that all events could arrive at the session-channel (assuming the client requires login).

The following illustrates some of the messages sent:

LOGIN request:
```
LOGIN /gold_session/ HTTP/1.1
Host:  test.daimi.au.dk
<a:login xmlns:a="http://www.daimi.au.dk/AwareDAV"
         xmlns:d="DAV:">
 <a:channel><a:tcp>
  <d:href>teds-computer.daimi.au.dk</d:href>
  <a:port>8001</a:port>
 </a:tcp></a:channel>
 <d:displayname>ted</d:displayname>
 <a:session-member/>
</a:login>
```

LOGIN response (creating a new session-channel):
```
HTTP/1.1 201 Created
Location:  test.daimi.au.dk/gold_session/ted
<a:login-response
    xmlns:a="http://www.daimi.au.dk/awaredav"/>
```

Open folder (PROPFIND and SUBSCRIBE requests):
```
PROPFIND /test HTTP/1.1
Host:  test.daimi.au.dk
Depth:  1
Session-channel:  test.daimi.au.dk/gold_session/ted
...
<d:propfind xmlns:d="DAV:">
  <d:prop><d:resourcetype/></prop>
</d:propfind>

SUBSCRIBE /test HTTP/1.1
Host:  test.daimi.au.dk
Depth:  1
Session-channel:  test.daimi.au.dk/gold_session/ted
...
<a:subscribeinfo xmlns:a="..." xmlns:d="DAV:">
 <a:what>
  <a:created/><a:deleted/><a:updated/><a:copied/><a:moved/>
  <a:bound/><a:unbound/><a:locked/><a:unlocked/>
 </a:what>
 <d:select><a:method/><d:resourcetype/></d:select>
 <a:channel>
  <d:href>test.daimi.au.dk/gold_session/ted</d:href>
 </a:channel>
</a:subscribeinfo>
```

SUBSCRIBE response:
```
HTTP/1.1 201 Created
Subscription-ID: 1051
```

And finally a notification (same as the body of a POLL response):
```
<a:notification-set><a:notification>
 <d:href>test.daimi.au.dk/gold_session/ted</d:href>
 <a:subscription-id>1051</a:subscription-id>
 <a:owned-by>test.daimi.au.dk/gold_session/ted</a:owned-by>
 <a:event>
  <a:what><a:created/><a:bound/></a:what>
  <d:prop>
   <a:method>MKCOL</a:method>
   <d:resourcetype>collection</d:resourcetype>
  </d:prop>
 </a:event>
</a:notification></a:notification-set>
```
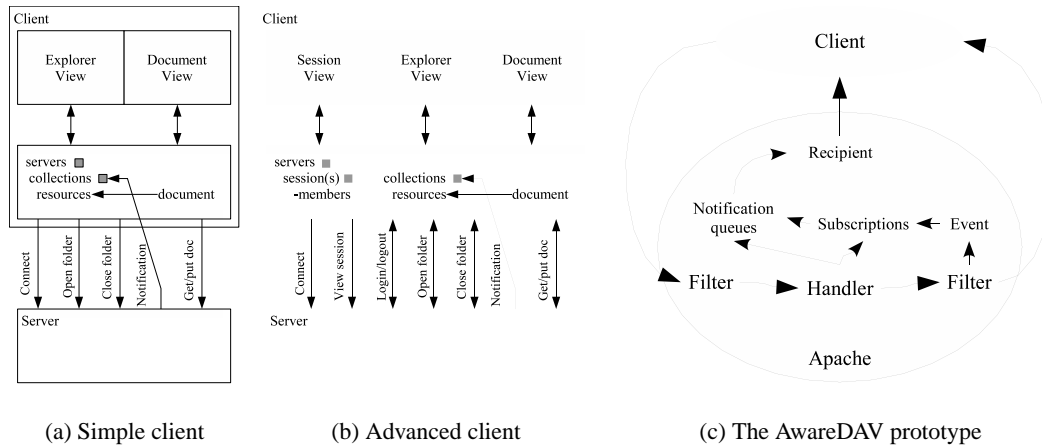
Having seen a few examples of client interactions, we describe our server implementation.

(a) Simple client   (b) Advanced client   (c) The AwareDAV prototype

Figure 2: API Usage Examples

7. IMPLEMENTATION EXPERIENCES

Any protocol or design must be validated through implementation. When faced with the implementation of AwareDAV, we decided to extend an Apache Web server (complete with mod_dav) with an AwareDAV module. This choice makes the extensibility of AwareDAV dependent upon the extensibility of Apache and the mod_dav module. Other choices could have been to build a server from scratch, or to choose a different Web server as platform.

Using the filter mechanism of Apache, we monitor all request arriving and all responses departing. Upon departure, we generate an event based on the information gathered and compares with known subscriptions. Matches cause a notification to be dispatched to the channel. Seven method handlers have been made to deal with the seven new request methods. Sessions-channels are just normal resources with the **a:session-channel** property, and a reference to the notification queue.

The protected properties (including **a:session-channel**) are implemented through a hook mechanism in mod_dav; apart from this, both the Apache Core and mod_dav are left untouched.

This approach gives some problems with the accuracy of the event information: either we have to make sub-requests, impairing efficiency, or we must accept that the events cannot provide full knowledge about what happened. E.g., the request/response of a PUT does not give enough information to return the **diff** element. Such a scenario is one of the reasons behind AwareDAV's differentiation between event-properties and resource-properties in the notification. Resource-Properties are explicitly retrieved after matching, while event-properties should be low cost and directly derivable from the event. Modifying mod_dav or using its hooks more extensively may alleviate this problem.

8. RELATED WORK

As mentioned in Section 3.2, AwareDAV is not the only framework for notification support in WebDAV. Notable are the support found in Microsoft Exchange and WebDAV-notify. These systems were sources of inspiration during development, and this section details the similarities and differences between AwareDAV and the two systems.

There is some common ground between AwareDAV and Microsoft Exchange [13]: Some of the request methods names (SUB-

SCRIBE, UNSUBSCRIBE, POLL) and their basic functionality, the unique subscription ID (which together with an URI identifies the subscription), inheritance of subscriptions, and a notification queue to hold the notification until polled.

However, there are also some key differences: There are only a few specialized event types in Microsoft Exchange, events use headers for all information, and that information can only be obtained through the POLL request. Our notifications are contained entirely in a XML body, so the only requirement to the notifying mechanism is that it is capable of delivering XML. Our design is far more flexible in these aspects, whereas the Microsoft solution has been adapted specifically to the mail server domain.

There are at least three major differences between AwareDAV and WebDAV-notify [12]: The data models, the event scheme and the server-gateway architecture.

- WebDAV-notify maps the WebDAV resource model unto XMPP-PubSub's node model. These two models differ in that collections in WebDAV may have both content and properties, while PubSub's collection are pure bindings. Especially in the more advanced WebDAV extensions (e.g., ΔV) the missing events for collection operations may be irritating.

- Clients can subscribe to node creation and item change events, and may therefore receive notifications that it should ignore.

  To distinguish between the notifications, WebDAV-notify change notifications are categorized by request method and payload elements. Though we could envision different methods having the same payload signaling similar behavior, this makes for a less adaptable design with regard to new extensions that use the same payload but should not be associated with existing methods.

  As our design supports human notification (like email), we sometimes need our finer grained matching mechanism. Even though it is possible to match for request names, it is not recommended in our design.

- WebDAV-notify has one primary communication conduit: XMPP. This has the advantage of encouraging both servers and clients to be more secure. Our design could probably also be adapted to XMPP-CORE, by simply having the server redi-

recting the subscription requests, as long as we are able to dispatch headers and XML bodies.

AwareDAV encourages gateways to make their existence known through WebDAV-properties along with all built-in channel communication types, while Webdav-notify assumes that the problem is handled by gateways and clients themselves.

These similarities and differences highlight the different design goals of the three projects. AwareDAV has the advantage of having a higher degree of flexibility and extensibility.

9. CONCLUSION

The introduction of WebDAV was an important conceptual shift on the Web, whereby the realization of the "inter-creative Web" came closer. WebDAV provides a wide-spread standard for basic collaboration support in the areas of uploading, locking, annotating, and versioning resources on Web servers. The common user will never directly come into contact with WebDAV, but will experience it as an enabling technology enabling work with remote colleagues.

AwareDAV supplements WebDAV in the important area of notification support, as this cannot be handled directly within the Web's strictly stateless client/server model. Staying within this model is a design feature of WebDAV, but constrains the ease with which collaborative systems can be built upon Web servers.

We have introduced AwareDAV as a new framework to support shared awareness through event notifications in a WebDAV based context. This allows for the creation of a new class of collaboration systems on the Web.

We have in this paper discussed two existing frameworks for notification support in WebDAV, namely WebDAV-notify and Microsoft Exchange. Both systems fit their niche well (and have indeed informed the design of AwareDAV), but are ultimately too restrictive for a general design. Microsoft Exchange can only push notifications through UDP and cannot provide users with exact information of what has changed; WebDAV-notify relies on XMPP for notification delivery, which may not fit all purposes, and in contrast to Microsoft Exchange provides the user with full change information (which may not always be necessary).

The AwareDAV approach is the more flexible as it allows for a range of delivery mechanisms, and provides a fine grained event subscription mechanism with an extensible event scheme to support future development. Any scheme related to collaborative work should be forward compatible to allow for new systems and innovative uses.

We have discussed our experiences with the initial implementation of AwareDAV on an Apache/mod_dav Web server. Our initial results are promising and the extensible approach taken by AwareDAV should allow for future growth and enable collaboration across the Web through the merging of WebDAV and notification systems.

Acknowledgment

10. REFERENCES

[1] K. M. Anderson and N. O. Bouvin. Supporting project awareness on the WWW with the iScent framework. *ACM SIGGROUP Bulletin*, 21(3):16–20, 2000.

[2] S. Bødker, M. Kyng, and K. Schmidt, editors. *Proceedings of the 6th European Conference on Computer Supported Cooperative Work*, Copenhagen, Denmark, Sept. 1999. Kluwer Academic Publishers.

[3] E. Bradner, W. A. Kellogg, and T. Erickson. The adoption and use of 'BABBLE': A field study of chat in the workplace. In Bødker et al. [2], pages 139–158.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, July 2000.

[5] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. Versioning extensions to WebDAV. Standards Track, Proposed Standard RFC3253, IETF, Mar. 2002. http://www.ietf.org/rfc/rfc3253.txt.

[6] G. Clemm, J. Reschke, E. Sedlar, and J. Whitehead. Web distributed authoring and versioning (WebDAV) access control protocol. Standards Track, Proposed Standard RFC3744, IETF, May 2004. http://www.ietf.org/rfc/rfc3744.txt.

[7] G. Fitzpatrick, T. Mansfield, S. Kaplan, D. Arnold, T. Phelps, and B. Segall. Augmenting the workaday world with Elvin. In Bødker et al. [2], pages 431–450.

[8] L. Fuchs. AREA: A cross-application notification service for groupware. In Bødker et al. [2], pages 61–80.

[9] L. Fuchs, U. Pankoke-Babatz, and W. Prinz. Supporting cooperative awareness with local event mechanisms: The GroupDesk system. In *Proceedings of the 4th European Conference on Computer Supported Cooperative Work*, pages 247–262, Stockholm, Sweden, 1995.

[10] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP extensions for distributed authoring – WebDAV. Standards Track RFC2518, IETF, Feb. 1999. http://www.ietf.org/rfc/rfc2518.txt.

[11] C. C. Heath and P. Luff. Collaboration and control: Crisis management and multimedia technology in London Underground line control rooms. *Journal of CSCW*, 1(1–2):69–94, 1992.

[12] J. Hildebrand and P. Saint-Andre. Transporting WebDAV-related event notifications over the extensible messaging and presence protocol (XMPP). Internet-draft, IETF, Sept. 2004. http://www.ietf.org/internet-drafts/draft-hildebrand-webdav-notify-00.t%xt.

[13] Microsoft. *MSDN: Exchange 2000 HTTP WebDAV Access*, Sept. 2004. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wss/ws%s/_exch2k_http_webdav_access.asp.

[14] W. Prinz. NESSIE: An awareness environment for cooperative settings. In Bødker et al. [2], pages 391–410.

[15] J. Reschke, S. Reddy, J. Davis, and A. Babich. WebDAV SEARCH. Internet-draft, IETF, Sept. 2004. http://www.ietf.org/internet-drafts/draft-reschke-webdav-search-07.txt.