

SAuthMash: Mobile Agent based Self Authorization in Mashups

Imran Khan
FAST-NUCES, Pakistan
imrancd9@yahoo.com

Mohammad Nauman
Security Engineering
Research Group
Institute of Management
Sciences, Pakistan
nauman@imsciences.edu.pk

Masoom Alam
Security Engineerin Research
Group
Institute of Management
Sciences, Pakistan
masoom.alam@gmail.com

Furqan Aziz
FAST-NUCES, Pakistan
furqan.aziz@nu.edu.pk

ABSTRACT

Mashups are web based applications that merge contents (data and code) from multiple sources, and provide an integrated view to the user. One of the main requirements in mashup is the authorization of user to the backend services. Current protocols for authorization in mashup have obvious limitations. With strawman approach a malicious or compromised mashup can leak user credentials. OAuth approach has the scalability problem and requires a statefull server at the backend service. AuthSub issues only single use token and obtaining session token requires additional steps and also explicit revocation, which may not be possible in some situation. The problem with Permit based approach is that it requires separate permit for each backend service and also require renewal or obtaining new permit in case of mashup requirements changes (e.g. read to execute). Revocation is a problem in this approach as well.

In this paper we propose a new protocol for accessing backend services in mashup. Our protocol makes use of Java based mobile agent called Aglet. The main source of problem in above approaches is due to delegating the authorization process to mashup. In our approach, mashup that require content from backend services that content is accessed and provided to the mashup through Aglet, without delegating authorization rights or releasing credentials to the mashup. Aglet has the ability to move around the nodes of a network and to sense its environment and to perform the desire actions. So the stated limitation of above approaches can be overcome with our Aglet based approach by allowing the Aglet to move across different mashup and backend services and provides data and code as necessary.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FIT'09, December 16–18, 2009, Abbottabad, Pakistan
Copyright 2009 ACM 978-1-60558-642-7-16/12/09 ...\$5.00.

Categories and Subject Descriptors

H.4.m [INFORMATION SYSTEMS APPLICATIONS]:
Miscellaneous

General Terms

Security

Keywords

Mashup, Authorization, Mobile Agents, Web.

1. INTRODUCTION

The ability of combing data and code from different sources is one of the main features of Web 2.0. A mashup is normally a web based application that combines content and functionality from a diversity of sources and provides an integrated view to the user. Mashups provides rapid method of creating a new web service through the combination of several different existing services on the Web.

Besides information retrieval, Web 2.0 technology allows the users to perform other interesting actions as well. Web 2.0 provides Network as platform and thus making it possible to execute software application through client browsers [12]. In Web 2.0 user has the capability to own data and to exercise control over that data. [1].

As mashup integrate content from multiple backend services, it needs to provide authorization information to the backend services. Currently there are many different protocol used for authorization in mashup. One approach is called all or nothing (strawman) approach. In this approach the user trusts the mashup by giving mashup her complete authorization credentials(usually username and password). If the mashup is malicious or compromised, it can lead to theft of user credentials.

There are also two authorization protocol, i.e. OAuth and AuthSub, from Google. In these protocols token are issued to the mashup to access backend services. The problem with OAuth and AuthSub is that tokens are single use and getting session token requires additional steps. Also the problem with OAuth is that it requires statefull server at the backend service to keep track of previously issued tokens, which doesn't scale well in case of large number of users.

A permit based protocol [8] is a new one and tries to solve the problem of previous protocols by making use of permit. In this approach user provides mashup with a permit to access backend service. However this approach has also some serious limitations. It requires separate permit for each backend service. Because mashup integrate content from multiple sources, it will be beneficial if a single entity or permit can be used to access content from multiple sources. If the access requirements of mashup changes (e.g. Read to Execute), it requires the renewal of the token. If a token is issued for a specified period of time, there is no guarantee that the token can be revoked before that time elapses. So revocation is one of the main issues in permit based protocol.

Hence, there is a need for an approach that is scalable and can support both statefull and stateless servers. Moreover it must support full revocation of granted permissions. Also different backend services shouldn't require separate permits or tokens so that a single entity or object can be used to access content from multiple backend services and to be delivered to the mashup. Also we need to give mashup only that contents that it requires for its operation. so the new approach must be able to filter the information to be given to the mashup. These requirements are not met in previous authorization protocols. Here we propose a protocol based on Java mobile agent, i.e. Aglet [3], which solves the above stated problems in mashup authorization protocols.

This paper is organized into 7 sections. In this section, we have provided an overview of the problem addressed in this paper. In section 2, we give a concise review of previous work done in mashup authorization along with its limitations. Section 3 provides a detailed statement of the problem, and Section 4 provides a comprehensive review of Mobile Agents Technology. In section 5 we have described in detail our protocol and explained its target architecture. Section 6 shows the achievement of our approach and Section 7 concludes the paper.

2. BACKGROUND

Mashup provides rapid method of creating a new web service through the combination of several different existing services on the Web [10]. Because creating a mashup from existing applications and services is much easier than creating a comparable service from scratch, mashups have proliferated on the Internet [11].

Mashups can be categorized as client side or server side, depending on where the mashups logic resides. In case of client side mashup [9] the contents from different site are mashed within the user browser. In case of server side mashup, content from different sites are combined at the web server, not in the client browser. Here our focus is on server side mashups. In this section we describe well known techniques that are used for authorization in mashup.

2.1 Authorization in Mashups

2.1.1 All or Nothing Approach

In this approach user first gives credential to the mashup site. The mashup then use these credentials to access the backend services. In this way the mashup impersonates the user at the backend services. This represents all-or-nothing approach, i.e. user either delegates all of the services to the mashup or none of them.

In this approach the user has to trust the mashup fully. The user gives all the privileges to the mashup, and the user has no option to restrict the mashup capabilities. In this way a compromised mashup can leak user credentials and can impersonate the user maliciously. Also there is no explicit revocation; the revocation is only possible if the user changes his/her credential manually at the backend service. In some situation the mashup allow the user to remove his/her credential from the system, but in that case the user has to trust the mashup fully.

2.1.2 AuthSub

AuthSub [2] was developed by Google to allow access to Google services. AuthSub requires the mashup to make a AuthSub call to the Google account URL, in order to access the user's Google service data. In response to user request the Google accounts responds with an "Access Consen" page. The page provides an opportunity to the user to log on to her Google account and to either grant or deny access to Google service. In case the user successfully logs in and allows access, Google account redirects the user back to the web application. The redirection also produces a single use authentication token. If required, the token can be exchanged for long lived one. The web application then uses the authentication token, contacts the Google service as a user agent. When authenticated, the Google service then provides the requested data.

AuthSub protocol from Google requires RPC API calls to validate tokens. The token contains no information about the backend service and the user to which it belongs. Deciphering a token require additional RPC calls. Here also the token is also single use. To get a session token additional RPC calls are required and in this case user has to revoke the token at the end of session. The revocation introduces many difficulty and is error prone and in some situation impossible. The AuthSub server must be statefull, in order to allow the users to renew or revoke already issued token. As a result the approach presents the scalability problem. To cater this problem, AuthSub can limit the token that can be issued at one time. So AuthSub specification doesn't permit more than ten valid token per user, per web application [2].

2.1.3 OAuth

OAuth [2] is a new protocol for authentication and allow distributed web application to get limited access to a distributed service. In OAuth an application that want to access backend services, first get an unauthorized token from the service. The unauthorized token is then given to the user. Also the user is redirected to the service provider. After getting user approval, the service provider converts the token into an authorized token, and redirects the user into consumer application. After getting the authorized request token, the consumer application uses it to get an access token. The consumer application can then use access token to get access to backend service. OAuth is a statefull approach; require the backend service to maintain the state of all previously issued tokens.

Compared with strawman approach OAuth is a better approach, and aims to solve the delegation problem. On the other hand, OAuth require statefull server at the backend service and issues token which are single use resulting in the scalability problem. Also OAuth approach needs dedicated resources and support. In OAuth corespec [4] no access re-

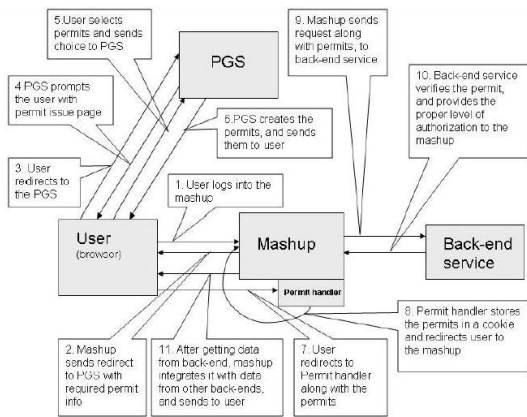


Figure 1: Permit Me [8]

restriction or scope is defined.

2.1.4 Permit Based Approach

Permit Based approach proposed in [8], is similar to real life permit based approach. Here is how permit based approach works in Mashup authorization. When the user send request to Mashup, the Mashup asks the user to provide permit to be used in accessing backend service for the user. The delegated permit is limited life time, unforgeable and a digital token to provide specified access to backend service. The user creates permit, represents permission, which the Mashup actually needs to access the backend service. The Mashup can use the permit, until the user revoke or the permit expires.

When the Mashup receives the permit it then sends the permit to the backend service with the request to access the backend service. After receiving the permit the backend service make decision based on the information contained in the permit. The user can review, revoke and modify all ready issued permits.

The problem with permit based approach is that it needs separate permit for each backend service. Also if the access requirement of the mashup changes, i.e. from read to write, needs issuance of new tokens. In some situation we may require stateless approach while in other statefull approach may be more suitable. However permit based approach [8] can only be used in stateless scenario.

The main problem with permit based approach is that revocation of token before its expiry is not possible. Referring to Figure 1, mashup keeps tokens issued to it by the user in permit handler, to be used for authorization in the future. Suppose, the user gives a token to a mashup to access a back-end service, whose expiry is 40 minutes. If after 10 minutes of token issuance the user decides to revoke the token from the given mashup, she cannot do so before it expires. The reason for it is that the mashup keeps the token in permit handler and it is up to that mashup whether to stop using the token or not.

3. PROBLEM DESCRIPTION

Currently well known techniques for authorization in mashup for accessing backend services are Strawman approach, OAuth, AuthSub and Permit based approach. There exist some serious deficiencies in all three of these approaches. Although

the Permit based Approach is a new one and aims to solve the deficiencies present in the previous approaches, it also has some serious limitations.

The techniques discussed in this paper for authorization in mashups for accessing backend services have the following deficiencies.

3.1 Delegation of Rights in Accessing Backend Services

In Strawman approach [8] the user gives his/her credential to the mashup to access backend services. So the user delegates all the rights to the mashup. Also in OAuth [4], AuthSub [2] and Permit Me, user delegates all its rights to the mashup for accessing backend services. So these approaches are not appropriate as they delegate rights to the mashup that the mashup doesn't requires. Permit based approach solve this problem by delegating only those rights that the mashup need at the backend service. But still delegates right to the mashup to access backend services. Now if the mashup is malicious or compromised, delegation of rights can be very harmful. So all of the previous approaches for authorization of backend services are vulnerable in case of a malicious or compromised mashup.

3.2 Authorization Requirement may Change Dynamically

As already discussed to overcome the deficiencies of Strawman, OAuth and AuthSub approaches to delegates only those rights that the mashup needs to access backend services, the Permit based approach only delegates those rights that the mashup needs. But if the access requirement of the mashup change, say from read to write, the permit based approach will need to renew the token or issue new token, which is very cumbersome and costly in terms of performance and complexity.

3.3 Problem of Separate Permit for each Backend Services

In Permit based approach, permit is issued for each back-end service. If a mashup integrate contents from ten backend services that need authorization, the user will need to issue ten tokens. So creating, renewing or revoking token can be cumbersome if each backend service requires a separate token. Therefore it is a main problem in Permit based approach to create a separate permit for each backend service. It would be better if a single entity, object or token can be used to access content from multiple backend services.

3.4 Problem with both Stateless and Statefull Approaches

Both statefull and stateless approaches have its own merits and demerits. The problem with statefull approach is that of scalability. In statefull approach the server has to keep track of all previously issued tokens. If the server has many registered users and they start accessing the server at the same time, it will be difficult for the server to keep record of all previously issued token. So the main issue with statefull approach is that of scalability.

Although the stateless approach solve the problem of scalability, it still has some limitations. In stateless approach subsequent access to backend service, in case of permit based approach, require to check with identity provider, to verify the permit, which is an extra overhead. So we need an ap-

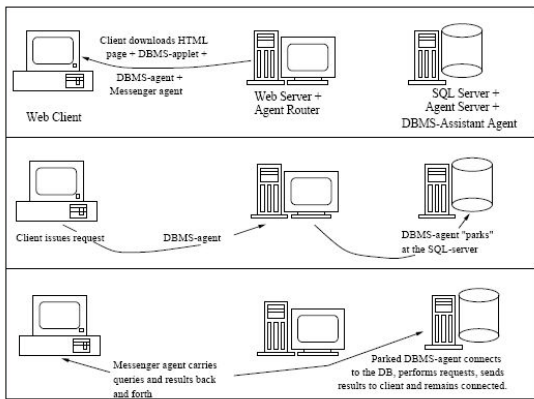


Figure 2: Aglet for Distributed Database Access [13]

proach which can support both statefull and stateless approaches, so that one can use it according to situation.

4. MOBILE AGENT

For addressing these issues, we envision the use of Mobile Agents [13] in mashup scenarios. Below, we provide a brief background on this aspect of our work.

Mobile Agent is a type of Software Agent, with the property of autonomy, social ability, learning, and most importantly, mobility. Mobile Agent has the ability to transfer its state from one host to another on the web, and to start execution from the saved state rather than from the beginning. Mobile Agent has the ability to choose to move to the next host, which is an evolved form of RPC. During transfer Mobile Agent saves its state on local machine and start execution from that state when it arrives on other machines on the web.

4.1 Aglet

Aglet [3] is a Java based autonomous mobile software agent. An Aglet doesn't need to start execution from the beginning on the new platform, it can start execution from the point where it has left execution. With Java Aglet technology both code and state is mobile [9].

Aglet are autonomous, meaning it can decide independently where it should go and what to do there. Aglet controls its life cycle itself. Aglet are able to receive request from external sources, but free what to do with these external requests. Aglet can perform action such as traveling across networks to a new host without any external request [3]. Aglets are designed around an event-driven programming model that has similarities with Applet programming. In the life time of Aglet the following events can occur: Creation, Disposal, Cloning, Dispatch, Retract, Deactivation and Activation. Before any of these events can be called, Aglet is first notified of these incoming events.

4.1.1 The Aglet Environment

Aglet communicates with its host environment through an AgletContext object. Using this object, Aglet accesses host services and resources, such as the local file system. Communication with another (local or remote) Aglet is possible through AgletProxy object of that Aglet. The AgletProxy provide a means of protecting it from malicious calls from other Aglet. The Agletproxy mechanism allows an Aglet to

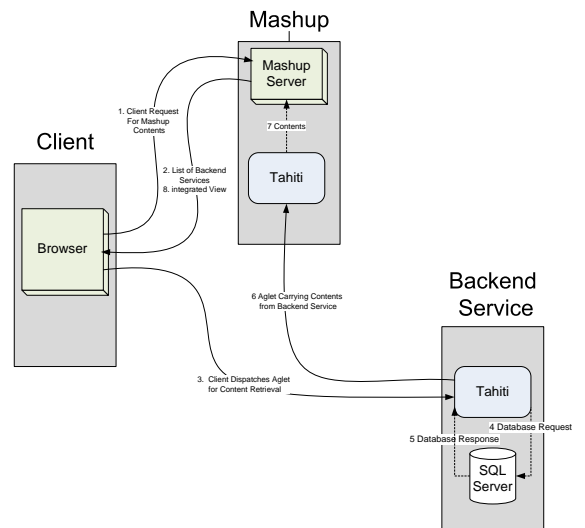


Figure 3: SAAuthMash

request an action of another Aglet, such as `dispatch()`, or `clone()`. The receiving Aglet is free to carry out the request, ignore it, or take some other action [6].

4.2 Aglet for Distributed Database Access

Aglet is already applied in distributed web applications such as Web Databases [13], Cooperative environment [5] and Information-gathering System [7].

In [13] Aglets are used to access distributed databases as shown in the Figure 2. In this model, the Aglet originates from the client machine, and is then dispatched to the SQL server and at resides there. Now it is the responsibility of the resident Aglet to perform action such as Database connectivity (JDBC connection), etc. Now another Aglet is used for communication between client and resident Aglet at the SQL Server.

After first request by the client, two Aglets are fired from the client machine. One of the Aglet resides at the SQL Server and performs client specified actions. The other Aglet acts as a communicator that carries messages between client and resident Aglet (at the SQL Server). Below, we describe an approach similar to this one for secure data access in mashup scenarios.

5. SAUTHMASH PROTOCOL

Here we describe the design and working of *SAAuthMash* – a secure authorization framework for mashup scenarios. The main entities involved in our protocol are Client machine, Mashup site, Database Server as backend service, and Aglet, as shown in Figure 3. There are eight main activates involved in the proposed protocol, starting from the client request and ending with mashup response with the desire contents.

Note that Aglet based approach is already used in distributed web database access [13]. Here through Aglet based approach we propose that how Authorization process in mashup can be improved from already proposed protocols.

Step 1: In step one the user sent normal http request

to the mashup, as shown in step 1, in Figure 3. As previously discussed, mashup combines contents from multiple sources and provide an integrated view to the user. Here the mashup logic resides at a separate site. There may be scenarios in which a server may provide mashup logic in addition to hosting its own contents, as is the case with Yahoo and Google servers.

As mashup integrate contents from other sources, some sources may release information without requiring any authorization, while others may need authorization in order to provide access to the content they host. Now when the mashup receives request from the client, it check to see what backend services need authorization information in order to provide its content.

Step 2: After mashup identifies the backend services it needs to retrieve contents from and which one needs authorization, it then provides a list of backend sources to the user which needs user authorization in order to provide its contents. The user is provided with a list of backend services for which authorization is needed to retrieve contents. Sources which require no user authorization in not mentioned in the list.

Step 3: Our protocol differ from other protocols in the way Step 3 through Step 7 are implemented. After receiving a list of sources which require authorization, the client makes use of Aglets to access information from the backend services. In existing protocols the user either gives its credential to the mashup to access backend service or delegates its right through token or permit based approaches.

Our approach makes use of Mobile Agent, i.e. Aglet, to avoid giving either credential or delegate rights to the mashup. The user will fire an Aglet from its machine to the backend service, so that it can retrieve information from that backend service. In cases where the user is provided with a list of sources, user has the option to send either a single Aglet or multiple Aglets to retrieve contents from those multiple sources.

In step three, Aglet is dispatched from the client machine to the backend service rather than to the mashup, (see Figure 3). The Aglet send to the backend service is a specialized Aglet used to represent the user there and to retrieve contents. User's authentication information is also given to the Aglet. The Aglet also contains logic to transfer the retrieved information to the mashup site. The transfer of Aglet involves both code and its state. So, on the received host it will start execution from where it left off on the client.

In case of stateless model, only a single Aglet moves between client, database server and mashup site. But in case of statefull model, one Aglet is permanently parked at the back end service and resides there. While another Aglet is used as a messenger agent to carry messages between client, parked Aglet and the mashup site.

Step 4: In step 3 Aglet is transferred from client to the backend service. Aglet only runs in its own execution environment. Tahiti server developed at the IBM Tokyo research laboratory provides an environment for Aglet execution. At the backend service the Aglet is received by the Tahiti server, as shown in the Figure 3. On Tahiti server the Aglet will be executed. From there data related request is generated for information retrieval and authorization.

Step 5: In this step, backend Service process data request received from Tahiti server. The server first performs authorization, to check the incoming requests and then the

desired action is performed. In this step, the result of Aglet request is then transferred to the corresponding Aglet which exist on Tahiti server.

Step 6: The Aglet then gets the relevant information from the backend Service, for which it has been sent from client machine to the backend service. If filtering is needed, the Aglet filters the information it has received from database server. Filtering is applied so that the mashup only gets the information it requires. The Aglet then decided where to move or transfer information next. As we have already mentioned, Aglet is an intelligent, autonomous mobile agent, that senses its environment and then acts intelligently. Aglet is sent to the backend service to retrieve relevant information, to filter it and then give it to the mashup. After retrieving, filtering information, the Aglet sent it to the Tahiti server on mashup site, to give mashup contents for integration.

Step 7: The mashup site will also host Tahiti server in order to process incoming Aglet. Now after accessing relevant information from the database server the Aglet will transfer itself to the Aglet environment on the mashup site, along with the retrieved information. The Aglet will execute there and will transfer the relevant content to the mashup site. A variation of step 7 is possible, which doesn't require the need of Tahiti server at the mashup site. After the relevant information from the database server is retrieved, rather than dispatching itself to the mashup site, the Aglet can generate a normal http response. In that response the relevant content will be transferred to the mashup site.

Step 8: Finally, the mashup receive content from the Aglet. For those backend services which don't require user authorization, the mashup will retrieve content from them directly. For those backend services which requires user authorization the mashup will get the contents through the generated Mobile Agent, i.e. Aglet. After getting contents from relevant sources the mashup will then integrate the contents, so as to provide an integrated view to the user. After combining contents, the mashup sends it to the client in this last step of our proposed protocol.

6. PROBLEM ADDRESSED BY OUR PROTOCOL

Our protocol addresses the problems mentioned in section 3. It has the following characteristics.

6.1 No Delegation of Rights to the Mashup

In our proposed protocol SAAuthMash, access remains under user's control and is not delegated to the mashup. When the mashup needs to access data from a backend service, Aglet accesses contents from that service; will then give it to the mashup for integration with other content. So the issues arising from the delegation of rights to the mashup avoided.

6.2 Support Change in Authorization Requirement with Low Overhead

To overcome the deficiencies of Strawman, OAuth and AuthSub, Permit based approach only delegates those rights that the mashup need. But if the access requirements of the mashup, say from read to write, change the permit based approach will need to renew the token or issue new token,

which is a very cumbersome and costly in term of performance and complexity. In case of our Aglet based approach, a single Aglet can be programmed to enable a mashup to retrieve or write information to a backend service. The Aglet fired from the user machine can be enabled to not only give the mashup the required information from the backend service but also can enable the mashup writing or execution capability at the backend services based on specific condition given to Aglet.

6.3 Support both Statefull and Stateless Model

As previously stated that the current protocols for authorization in mashup are either statefull or stateless. Both statefull and stateless approaches has its merits and demerits. It will be better to provide both option in one protocol and allow one to be chosen based on requirements. In case of our protocol both statefull and stateless option can be selected by the Aglet.

6.4 Protection against Man in the Middle Attack

In strawman approach, the user releases her credential to the mashup and if the mashup is malicious or compromised, man-in-the-middle attack involving the theft of user credentials is possible. In SAAuthMash approach, no credentials are released to the mashup, man-in-the-middle attack is not possible.

6.5 Filtering Information

In additional benefit of Aglet based approach is that filtering of information is possible. After the Aglet access information from backend service, it can filter information accessed from the backend service and can only release information to the mashup that is relevant. The mashup only view information given by the Aglet and actual structure of information is now hidden from the mashup. In previous approaches this kind of filtering was not possible.

6.6 Access of Information from Multiple Sources

In previous approaches each individual backend service needed a separate token or permit for each backend service. So the token management at the mashup site becomes very complex. In our proposed, protocol a single Aglet can be used to access information from multiple sources. This is possible because Aglet is free to move around the web and hence the same Aglet can be used to access information from multiple sources by visiting them in turn or through cloning.

7. CONCLUSION

Authorization is one of the main security concerns involved in mashups. Existing protocols for authorization to backend services delegate authorization to mashup which leads to many problems. In this paper, we proposed a new protocol for accessing backend service in mashup. Our Aglet based protocol avoids delegating authorization rights to the mashup, but at the same time provides contents to the mashup for integration. It avoids the short coming of previous protocols through the use of Mobile Agent, i.e. Aglet.

8. REFERENCES

- [1] Dion Hinchcliffe's Web 2.0 blog. Available at: web2.socialcomputingmagazine.com.
- [2] Google. Google Account Authentication (AuthSub). Available: <http://code.google.com/apis/accounts/AuthForWebApps.html>.
- [3] IBM, Aglets Software Development Kit, Aglets Documentation. 2005. Available: <http://www.trl.ibm.co.jp/aglets/>.
- [4] OAuth Specification 1.0. 2007. Available: <http://oauth.net/core/1.0>.
- [5] A. Castillo, M. Kawaguchi, N. Paciorek, and D. Wong. Concordia as enabling technology for cooperative information gathering. In *Japanese Society for Artificial Intelligence Conference*, pages 228–237, June 1998.
- [6] Nick Craswell, Jason Haines, Brendan Humphreys, Chris Johnson, and Paul Thistlewaite. Aglets: a good idea for spidering ? Available: research.microsoft.com/pubs/65286/craswell_idea97.pdf.
- [7] M. Dikaiakos and D Gunopoulos. The architecture of an internet based financial information gathering infrastructure. In *Proceedings of the International Workshop on Advance Issues of E-Commerce and Web-based Information Systems, IEEE Computer Society*, Apr 1999.
- [8] Ragib Hasan, Marianne Winslett, Richard Conlan, Brian Slesinsky, and Nandakumar Ramani. Please permit me: Stateless delegated authorization in mashups. In *Proceedings of the Annual Computer Security Application Conference, IEEE Computer Society Press, Anaheim, California*, Dec 2008.
- [9] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. Mashupos: Operating system abstractions for client mashups. In *HotOS*, 2007.
- [10] N. Kulathuramaiyer. Mashups: Emerging application development paradigm for a digital journal. *Journal of Universal Computer Science*, 13(4):531–542, Apr 2007.
- [11] D. Merrill. Mashups: The new breed of web app. 2006. Available: <http://www.ibm.com/developerworks/xml/library/x-mashups.html>.
- [12] Tim O'Reilly. What is web 2.0. *O'Reilly Network*, Aug 2006. Available: <http://www.oreilly.de/artikel/web20.html>.
- [13] S. Papastavron, G Samaras, and E Pitoura. Mobile agents for www distributed database access. In *Proceedings of the Fifteenth International Conference on Data Engineering*, pages 228–237, Mar 1999.