2010 34th Annual IEEE Computer Software and Applications Conference

Key Roles of Session State: Not against REST Architectural Style

Takeru Inoue*, Hiroshi Asakura†, Hiroshi Sato*, Noriyuki Takahashi*
*NTT Network Innovation Laboratories
Yokosuka-Shi, Kanagawa, 239–0847, Japan
Email: takeru.inoue@ieee.org
†Innovative IP Architecture Center, NTT Communications Corporation
Minato-ku, Tokyo, 108–0023, Japan

***Abstract*—The modern Web architecture basically follows the Representational State Transfer (REST) style. This style offers the architectural properties necessary to implement the Internet-scale Web. However, most authentication and delegation technologies that rely on *session state* actually deviate from the REST style. It must be noted, however, that the diversity of these technologies is imperative for the success of the Web.**

In this paper, we make a detailed analysis of current authentication and delegation technologies including OpenID and OAuth as well as HTML forms and cookies, and find that session state has an important role in terms of the diversity of the technologies. We also clarify that the negative impact of session state on the REST style is rather limited. On the basis of our analysis, this paper introduces the REST Using Session (RESTUS) architectural style, which is an extended REST style for sessions; this style places a session constraint on component interactions and so induces some properties required for the diversity.

***Keywords*-architectural style, REST, Web**

I. Introduction

A software architectural style [1] is basically a coordinated set of architectural constraints that restrict the roles and features of architectural elements, and the allowed relationships among those elements. When designing an architecture, software architects select a style to guide the design process since the style formalizes the key architectural properties required. Modern Web elements, such as HTTP and URI, were designed around the REpresentational State Transfer (REST) architectural style [2]. The REST architectural style places constraints on component interactions to induce the architectural properties required for the success of the Internet-scale distributed hypermedia system, namely, the Web. In the REST style, interactions are constrained so as not to depend on any *context* stored on a server. This constraint induces the property of scalability, because all interactions are independent of each other and so can be processed by multiple servers in parallel.

Contexts are, however, stored on innumerable Web sites. The stored contexts actually violate REST's constraints, though they are considered to be a necessary evil for permitting a user to advance through a set of successive interactions. In this paper, the set of successive interactions processed under a consistent context is called a *session* (this makes us call the stored contexts a *session state*). Sessions are usually maintained by using cookies [3], [4] as follows. After authenticating a user (e.g. validating user's password), the server stores the user's context together with a session identifier. The session identifier is set in the cookie of the response, which is transferred to the client. The cookies are automatically attached to subsequent requests by the client. The server processes the client's requests in the user's context.

The emerging authentication and delegation technologies of OpenID [5] and OAuth [6] are gathering much attention. In these technologies, components are allowed to establish sessions among more than two components. By verifying a trust chain among them, new sessions are established without prior sharing of credentials (e.g. a password). OpenID provides the single sign-on feature; users who have been authenticated by the authentication server, can establish sessions with other servers. OAuth allows users to grant their access authorities to servers, which use the granted authorities when establishing new sessions with other servers. Several large Web sites including Google and Yahoo! have already introduced these technologies, because they are essential for permitting modern Web sites to interwork with each other by flexibly establishing sessions.

It is clear that sessions play a key role in the Web, however, HTTP has no standard for session management (cookie is not a standard header for HTTP, nor is it designed to contain session identifiers). This is because session state violates the REST constraint. We need to rethink the REST style to answer the following questions on session state: *Why do we find so many session states on servers despite their violation of the REST style? What roles does it play in practice against the REST style? Does session state actually have a negative impact on the REST style?* Unfortunately, the session state has not been discussed well in the literature, and is not well understood. Answering these questions will give us deep insights for understanding and improving the authentication and delegation technologies for the Web.

This paper discusses the session state stored on Web servers. Our analysis of several session management schemes finds that session state has an important role in introducing a variety of authentication and delegation technologies. Our analysis also clarifies that the negative impact of session state is actually

0730-3157/10 $26.00 © 2010 IEEE
DOI 10.1109/COMPSAC.2010.64

rather limited. We propose an architectural style called RESTUS, which stands for Representational State Transfer Using Sessions. The RESTUS style places a session constraint on component interactions. The session constraint induces several architectural properties that support diverse authentication and delegation technologies, from HTML forms and cookies to OpenID and OAuth.

The remainder of this paper is organized as follows. Section II examines authentication and delegation technologies to clarify key roles of session state. Section III discusses the negative impact of session state on the REST style. Section IV proposes the RESTUS style based on our discussion. Section V details related work, and finally Section VI summarizes this paper.

II. Session Management Schemes in Current Web

This section examines the authentication and delegation technologies widely used in the current Web: HTML forms and cookie sessions, URI sessions, and OAuth. OpenID is briefly discussed in this section together with forms and cookies (it will be more fully described in Appendix A). We also discuss the key roles of session state.

A. HTML Forms and Cookie Sessions

Almost all HTTP authentication that involves a human using a Web browser is accomplished through HTML forms, with session identifiers contained in cookies. HTML forms provide a large degree of control over presentation, which is imperative for many Web sites. For cookies, most implementations rely on the *Netscape specification*, which is described loosely in Section 10 of RFC 2109 [3], though it was obsoleted by RFC 2965 [4].

An example of HTML form authentication and cookie session is shown in Fig. 1. Joe enters his username and password into a log-in form provided by the server (Fig. 1. 1-1). This information is transferred in the request (Fig. 1. 1-2), and authenticated by the server. If the authentication succeeds, the server generates a *session identifier* and stores it together with Joe's context (see the session table in Fig. 1). A stored session identifier associated with a context is called a *session state*. The session identifier is set in the Set-Cookie header in the response, which is sent to the client (Fig. 1. 2-1). The client is forced to set the session identifier in the Cookie header of all subsequent requests sent to the server (Fig. 1. 2-2). The server extracts the session identifier from the requests and finds the associated context. Finally, the requests are processed in Joe's context.

This framework is also used in OpenID. The detailed OpenID interactions are described in Appendix A.

B. URI Sessions

In some implementations that do not support cookies, e.g. i-mode 1.0 [7], session identifiers are set as a query string in the requesting URI. We illustrate this approach in Fig. 2. First, the server authenticates user Joe by using HTML forms and stores the session state on itself, in the same manner as

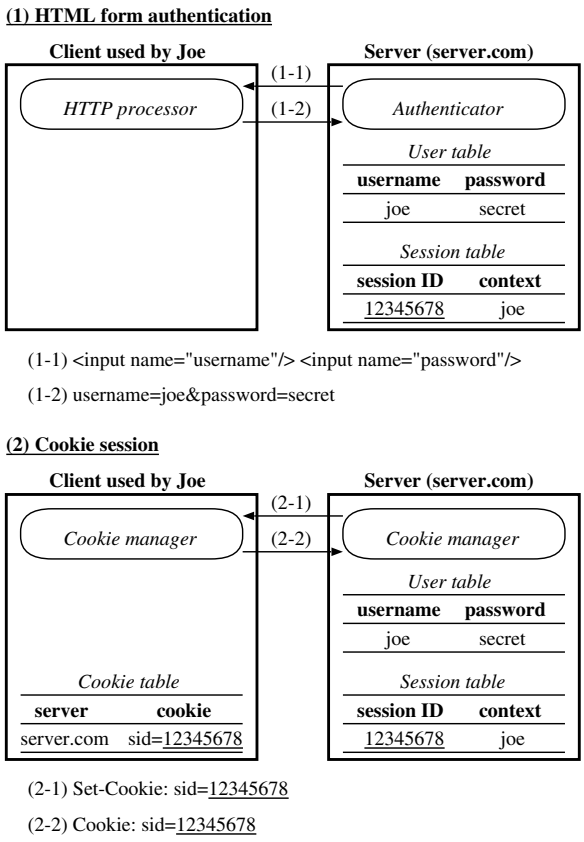

Fig. 1. An example of component interaction using HTML forms and cookies; step (1) is followed by step (2).

described in the previous subsection. Next, the server guides the client to URIs including the session identifier, by returning a response with a Location (redirection) header field or links to the URIs (Fig. 2. 2-1). As a result, the client requests the URIs of this session (Fig. 2. 2-2). The server, finally, extracts the session identifier from the requesting URIs, and processes the requests in Joe's context. In this way, sessions can be established between clients and servers by using URIs.

C. Sessions in OAuth

OAuth [6] enables servers to access protected resources on another server, without requiring users to disclose their credentials. In other words, the user safely grants his access authority to a server that then establishes a new session with the resource server. In OAuth, the server that maintains the resources is called a service provider, and the other servers that acquire the access authorities are called consumers.

Figure 3 shows an example of the delegation process in OAuth. For the sake of simplicity, we omit the procedures used for sharing a secret and exchanging unauthorized tokens. First, a client makes a request for the consumer's service that requires protected resources on the service provider (Fig. 3. 1-1). The consumer redirects the client to the service provider (Fig. 3. 1-2, 1-3). The service provider authenticates user Joe,

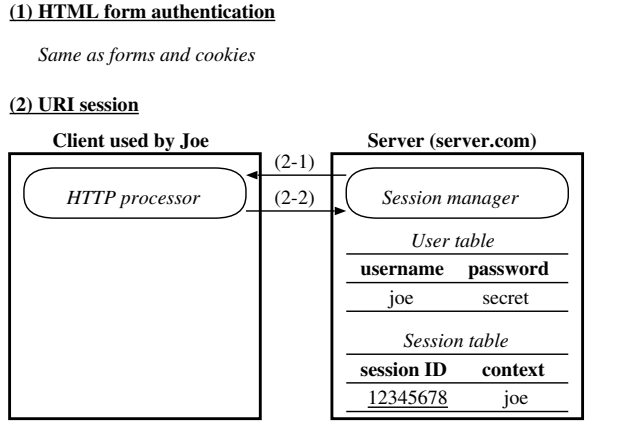

Fig. 2. An example of component interaction using HTML forms and URIs; step (1) is followed by step (2).

for example by using HTML forms (Fig. 3. 2-1, 2-2), if he has not already been authenticated (a session exists between the service provider and the client if he has been authenticated). The service provider generates a session identifier, called a token in OAuth, and stores it together with Joe's context (shown as the session table in Fig. 3). The service provider asks Joe whether he will grant his access authority to the consumer (Fig. 3. 2-3). If approved (Fig. 3. 2-4), the client is redirected back to the consumer (Fig. 3. 3-1, 3-2); this redirecting URI includes the session identifier (shown as the oauth_token parameter in Fig. 3), which is e-signed by the service provider. If the e-sign is successfully validated, the consumer asks for the protected resources using the session identifier (Fig. 3. 3-3). In this way, sessions can be established between consumers and service providers.

D. Key Roles of Session State

As described above, there are three types of sessions found in the current Web: cookie, URI, and OAuth sessions. Why were they introduced to the Web, even though they violate the REST style? We find that the two key roles of session state are as follows.

1) Decoupling Authentication with Session Management: The first key role is that session state makes the authentication process independent from the session management process (here, processes in advance of sessions are simply called authentication processes, though they possibly include delegation as seen in OAuth). In common among the technologies shown in Figs. 1–3, authentication results are carried over to the following session management process by using the session state. The authentication process is not tightly coupled with any single session management process, and vice versa. This means that these processes are replaceable; for example, the form authentication can be followed by URI sessions as well

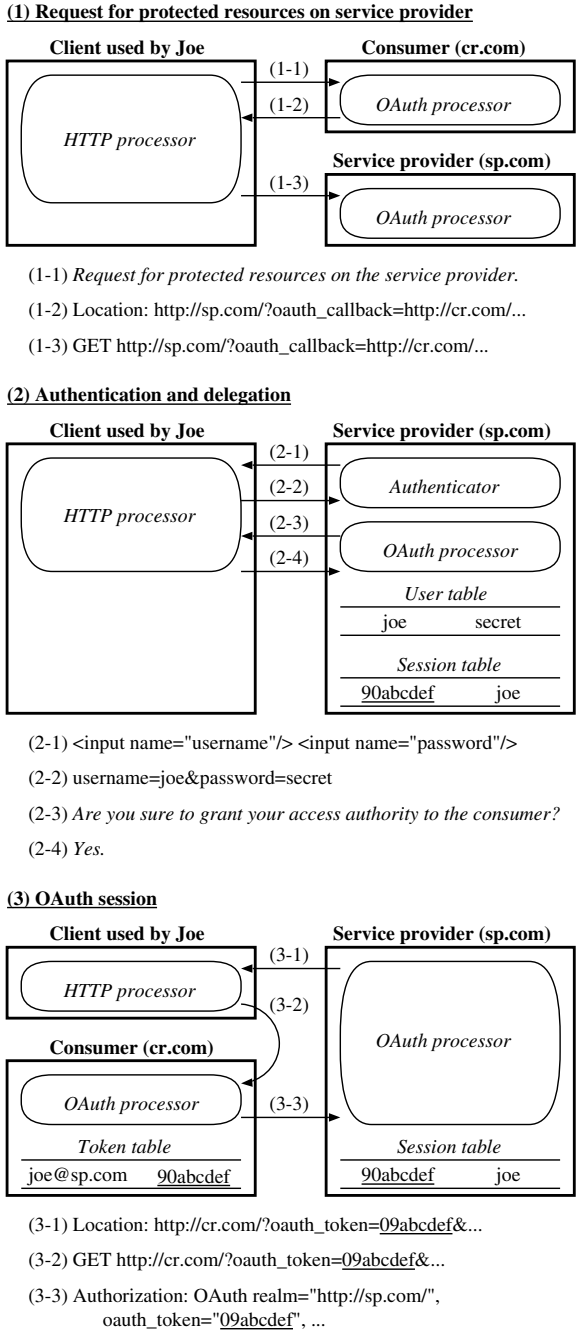

Fig. 3. An example of component interaction in OAuth; step (1) is followed by steps (2) and (3).

as cookie sessions.

In contrast, in the access authentication framework, which is provided by HTTP [8] and follows the REST style, the authentication process and session management process are integrated into a single scheme. For example, introduction of the form authentication into this framework requires some modification on the framework; the form semantics have to be

described in the authentication header, e.g. *field names for user and password are "usr" and "pswd", respectively*, in order to make clients understand them and thus create the appropriate response value [9].

2) Maintaining Sessions: Of course, session state is also required for maintaining sessions. The technologies discussed in this section have a common architectural pattern, as shown at the bottom of Figs. 1–3. This pattern represents session management; a server maintains a session state, which is a list of session identifiers associated with contexts, and a client sends requests with his session identifier.

Session state provides an identifier for each session. This identifier should be an opaque value (e.g. a hash), not a username or credentials. To prevent collision attacks and request forgery, malicious attackers must be unable to predict the identifier. In OAuth, an identifier (a token) is transferred via a consumer, and so must be opaque even if secure channels are used.

III. Impact of Session State on REST Style

This section reviews the REST style and discusses the impact of session state on the REST style. The discussion shows that session state places a negative impact on some properties (roles) of the REST style, but it is not significant.

A. Review of the REST Style

The central characteristic of the REST style [2] is interface uniformity. The uniform interface decouples implementations from the services they provide, which encourages the independent evolution of components. This is why so many implementations have been developed in the Web. With regard to the uniform interface, the key abstraction of information is a resource identified by a URI. A resource can be a document, an image or a service, which is exchanged as a representation, such as HTML, XML, JPEG, or any other digital format.

The REST style has two primary types of components, clients and servers; a client requests a resource, and a server returns a representation of the requested resource. Interactions in the REST style are stateless in nature. Each interaction is independent of other interactions, since each request contains all of the information necessary to understand it. As a consequence, servers are not required to retain any state between requests. This stateless constraint induces the properties of scalability, reliability, and visibility as follows. Scalability is improved, because requests can be processed by multiple servers in parallel thanks to their independency[1]. Reliability is also improved, because the failure of a single server causes no state loss and service recovery is simplified. Finally, visibility is improved, because a monitoring system does not have to look beyond a single request to understand it.

In the REST style, responses of interactions can be cached thanks to the cache constraint; this constraint requires that responses be labeled as cacheable or noncacheable. Since caching has the potential to eliminate several interactions, efficiency can be improved. Caching is performed by intermediaries as well as clients; caching by intermediaries is called shared caching. Interactions must be visible to (understood by) the intermediaries for shared caching.

We focus only on the stateless and cache constraints, even though the REST style has other constraints. This is because only these constraints are impacted by session state.

[1]Reference [2] explains in a different way why scalability is improved; it is because no state is retained and implementation is simplified. In reality, scalability is mostly improved by scaling out due to the parallelism of the current Web.

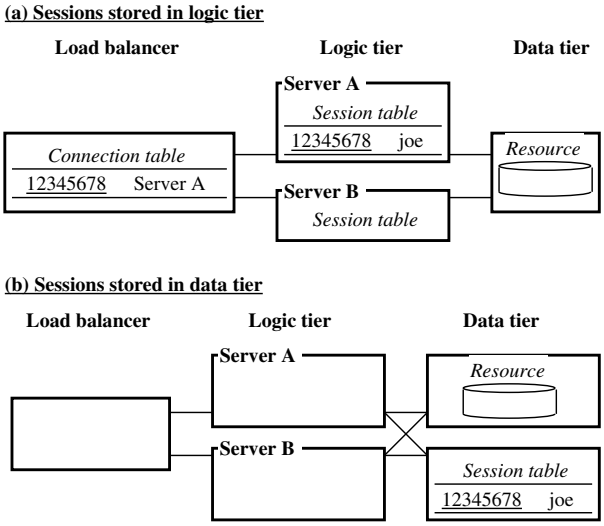

Fig. 4. An example of typical multi-tier servers.

B. Impact on Scalability and Reliability

As described above, the stateless constraint of the REST style induces the properties of scalability, reliability, and visibility. Unfortunately, session state negatively impacts these properties as follows. Scalability is degraded, because the load balancer is forced to maintain a connection state for distributing requests to appropriate servers as described below. Reliability is also degraded, because sessions are lost with server failure. We focus on reliability and scalability here; visibility is discussed in the next subsection as an issue of shared caching.

We explain how session state affects the properties of the REST style using Fig. 4 (a), which depicts the example of a typical multi-tier server [10]. The load balancer distributes requests to application servers in the logic layer. The data layer consists of database servers that manage resources. We assume that when user Joe requests authentication, the request is dispatched to Server A, one of the application servers. Server A authenticates Joe and instantiates a session state for Joe. Server A returns a response with the session identifier to the client. Here, the load balancer snoops the response to maintain the connection state, which is a list of session identifiers and corresponding servers [11]. The load balancer finally distributes requests according to the connection state. Clearly, the failure of Server A causes loss of the session state,

which causes a session break. Furthermore, the complexity of this load balancing mechanism reduces efficiency.

Our contention is that these effects are, however, not significant. In advanced Web sites, session state is stored in the data layer [12], as shown in Fig. 4 (b). This is due to the great advances made in the scalability and reliability of modern database technologies [13], [14], [15]. In this architecture, the failure of an application server no longer has any impact on the session state. Moreover, the load balancer is free from the need to manage connection state, since the session state is shared among all application servers.

C. Impact on Shared Caching

In interactions through sessions, the context is invisible to intermediaries, because it is confined to a server as session state. Therefore, responses of session interactions are not cacheable by an intermediary cache server, namely, shared cache.

Nevertheless, caching is still effective for the entire system, for the following reasons. First, responses without a session are cacheable in the same way as is possible with the REST style. Even if a response involves a session, it can be cached by a nonshared cache, namely, the client itself. Finally, regardless of visibility, session responses are inherently not cacheable by a shared cache, since their contents differ with the session.

IV. RESTUS Architectural Style

This section proposes the architectural style called RESTUS, which stands for REpresentational State Transfer Using Sessions. The RESTUS architectural style builds upon the foundations offered by the REST style, and offers further tools for maintaining sessions. We first focus on the essential architectural properties of sessions, on the basis of its key roles discussed in the previous sections. The architectural elements and views are then presented, and finally the constraints on those elements to achieve the desired properties are discussed.

A. Architectural Properties

We discuss below several architectural properties related to the essence of sessions.

1) Independency: Independency in the RESTUS style is defined as the degree of decoupling between authentication processes and session management processes. Improving this property is the main motivating force in this paper, since it facilitates service diversity. For example, this property brings presentational control to authentication processes, while the access authentication framework that follows the REST style does not provide it, as discussed in Section II-D. Presentational control is an important feature for human interfaces, even in authentication processes, since it controls the look and feel, and presents images and movies. Moreover, image selection (e.g. *which photo do you prefer?*) is recently used as an alternative authentication method.

2) Identity: Identity is defined as the ability of a system to identify and classify interactions based on authentication results. The interactions should be identified by servers at least, and possibly by clients and third parties. Authentication processes are not limited to pairwise interactions, because service diversity is greatly improved by introducing a third party to the processes, as shown in OpenID and OAuth. Session identifiers may cover new authentication or delegation processes in the same way as delegation processes are identified by tokens, session identifiers, in OAuth.

3) Scalability: In distributed environments, scalability is defined as the degree with which the system can handle an increasing number of components. A scalable architecture is easily configured to serve any number of client requests. There are two primary ways to improve scalability; first, by scaling up the system, that is to add resources to a single server, and second by scaling out the system, that is to add more servers to the system. In general, the scale out strategy is preferable, because of its cost effectiveness. The REST style fits the scale out strategy, since requests can be processed in parallel due to the stateless constraint. This property is maintained in the RESTUS style, as shown in Section III-B.

4) Reliability: This paper defines reliability as the ability of a system to perform its required functions under stated conditions for a specified period of time. The REST style improves reliability by applying the stateless constraint. This property is maintained in the RESTUS style, as shown in Section III-B.

5) Visibility: Visibility is determined by the degree to which an external mediator is able to understand the interactions between two components [2]. The easier it is for the mediator to understand the interactions, the more visible is the interaction between those two components. This property is required for shared caching. As discussed in Section III-C, this property is lost in the RESTUS style, but caching remains effective for the entire system.

B. Architectural Elements

Following [1], the key architectural elements of RESTUS are divided into three categories: processing, data, and connecting elements. The elements are shown in Fig. 5 and explained in the next subsection.

1) Processing Elements: Processing elements are defined as those components that transform the data elements.

Authentication processors are placed at *clients* as well as *servers*. They can be also set at *third parties* if the third parties are involved in the authentication process. The authentication processors are coordinated to determine the context stored on the server for the subsequent interactions. Actual behavior of the authentication processors are not constrained by the RESTUS style, hence there can be several approaches to determining the context, from simple client-server authentication to delegation of authorities.

Session managers are placed on clients and servers and coordinated to maintain sessions. The session manager at server-side is responsible for the management of session state

(the session table at the server in Fig. 5), which is a list of session identifiers associated with contexts determined by the authentication processors. The session identifier is generated by the server-side manager, and transferred to the client-side manager; it can be transferred via third parties, as shown in OAuth. When the server-side manager receives a request, it finds the context associated with the session identifier in the request.

The session manager at client-side may maintain session state (the session table at the client in Fig. 5). This session state is, however, slightly different from that at the server-side; it is a list of session identifiers received from servers together with server names. The client-side manager sets this session identifier in requests sent to the corresponding server, if the server is found in the session state.

Intermediaries, which are components placed between clients and servers, must be transparent for sessions. They are not shown in Fig. 5.

2) Data Elements: Data elements contain the information that is used by and transformed by the processing elements.

Contexts are determined by the authentication processes and stored in the session state. The interactions that follow the authentication processes are influenced by the contexts. Representation of contexts is not limited to an identifier of authenticated user; it can be a hash value of the user identifier or a set of resources to be accessed.

Session identifiers are keys to the corresponding contexts stored in the session state. A session identifier is an arbitrary bit-string, which is usually a hash value calculated by using a representation of a context, date-time, and/or random numbers. Session identifiers must be long enough to avoid collision attacks. Session identifiers can be e-signed to prevent forgery, if clients and servers share a secret. To avoid replay attacks, session identifiers should be replaced periodically. The same session identifiers should not be reused, to do otherwise would trigger cross talk.

Session State at server-side (the session table at the server in Fig. 5) is a mapping between session identifiers and corresponding contexts. That at client-side (the session table at the client in Fig. 5) is a mapping between servers and session identifiers. The client-side session state is optional (it is not used in Fig. 2).

3) Connecting Elements: Connecting elements serve as the glue that holds the components together by enabling them to communicate.

Session forms the basis of the interaction model in the RESTUS style. It presents an abstract interface to communicate in a consistent context. A server-side connector provides session identifiers, and validates them when received from clients. A client-side connector returns the session identifiers to the server, thereby maintaining the session.

C. Process View

Given the processing, data, and connecting elements, we can use different architectural views to describe how the elements work together to form an architecture. Here, we use a processing view, i.e. we concentrate on the data flow and some aspects of the connections among the processing elements with respect to the data [1].

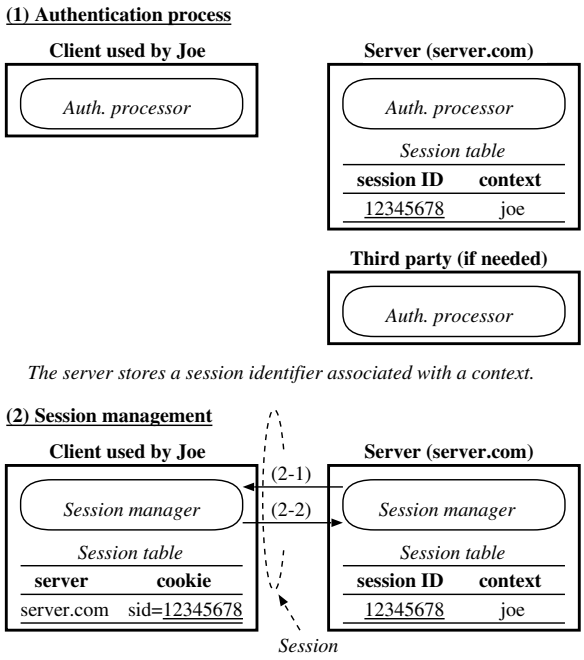

Fig. 5. Processing view of a RESTUS-based architecture; step (1) is followed by step (2).

Figure 5 depicts the processing view of an RESTUS-based architecture based on run-time component interaction. Figure 5 (1) indicates that there is an authentication process that creates a session state on a server. The process possibly involves a third party. The server authenticates the user, whether directly or not, to determine the context to be stored. The server, then, generates a session identifier, and stores it together with the determined context.

Figure 5 (2) illustrates an interaction for exchanging the session identifier between the client and the server. The server sends the session identifier to the client, either directly or indirectly. The client then sets the session identifier in the subsequent requests sent to the server. Finally, the server finds the context corresponding to the session identifier, and processes the requests according to the context.

The authentication process is decoupled with the session management process, since the authentication results are carried over to the session management process through session state. Moreover, session state enables servers to identify sessions. In this way, session state induces the properties of independency and identity.

D. Architectural Constraint

Architectural constraints can be used as restrictions on the roles of the architectural elements to induce the desired architectural properties in a system. We add a new constraint called *session* to the REST interaction; contexts of interactions are specified by session identifiers, and they are stored in a session state on a server. The session constraint retains the properties of independency and identity as discussed above.

Session state, however, violates the stateless constraint of the REST style. Nevertheless, since the stateless approach limits the diversity of authentication processes, our stateful solution is reasonable with minor losses in scalability, reliability, and visibility.

V. Related Work

There is some research work that extends the REST style. SPIAR [16] and CREST [17] are architectural styles for Ajax, which is a development approach for interactive Web applications. While SPIAR induces the property of interactivity that is essential for Ajax, it does not yield any of the properties desired for sessions. CREST extends resource representations into computational continuations, which encompass sessions. A continuation is, however, such a broader range concept that this style does not induce any specific property like a session. ARRESTED [18] is another architectural style based on the REST style. This style introduces several constraints, one of which, the route constraint, is slightly similar to OAuth. However, the route constraint focuses on a trust chain rather than a session.

Some papers have investigated architectural styles or patterns for secure communications. Reference [19] proposed an architectural pattern which describes pairwise authentication, encryption, and key exchanges. However, this style does not include sessions explicitly. Single sign-on and delegation of authorities also lie outside its scope. Dual Protection Style [20] is another architectural style for securing software, but its scope is limited to access control. Reference [21] describes an architectural pattern named session. However, this session is completely different from that of the RESTUS style; it is a software object that coordinates multiple accesses to a shared object.

The protocols of Grid were designed on the basis of the security policy defined in [22]. While the Grid and the Web have substantially different architectures, they have much in common. For example, two Grid protocols, the resource allocation protocol and the resource allocation from a process protocol, expand communication topologies such as OpenID and OAuth, respectively. We believe that the RESTUS style will play important roles in the Web, as does the security policy in Grid.

There are several identity management technologies that involve sessions. XRI and XDI [23] provide an abstract naming layer on HTTP, but sessions are beyond their scope. The Liberty Alliance [24] has standardized some identity management protocols based on SOA [25], namely, the "Big" Web services technology stack [26]. However, since no architectural constraints are introduced in SOA [27], these protocols are not discussed in this paper.

VI. Conclusions

This paper discussed the session state stored on Web servers. We clarified the key roles of session state (independency and identity), and defined them as architectural properties. We also revealed that session state is not actually hostile to the REST style.

In this paper, we first made a detailed analysis of several authentication and delegation technologies including OpenID and OAuth as well as HTML forms and cookies. Our analysis revealed that session state is required to decouple authentication processes from session management processes. This decoupling yields the flexibility of authentication processes. We also examined the negative impacts of session state and clarified that they are limited. On the basis of the analysis, we introduced the RESTUS style; this style introduces the session constraint on component interaction to induce the desired architectural properties.

While sessions have a key role in the current Web, they have not been investigated well from the perspective of software architecture. We believe that our work offers deep insights on the future of the authentication and delegation technologies in the Web.

Appendix

A. OpenID

OpenID [5] enables users to establish sessions without prior sharing of users' credentials, by introducing a dedicated authentication server. In OpenID, the authentication server is called an OpenID provider, and the servers with which the users establish sessions are called relying parties.

Figure 6 shows an example of log-in procedures in OpenID. For the sake of simplicity, the steps used to share a secret between an OpenID provider and a relying party are omitted. First, a client visiting the relying party selects its OpenID provider (Fig. 6. 1-1). The relying party then redirects the client to the selected OpenID provider (Fig. 6. 1-2, 1-3). The OpenID provider authenticates user Joe, if he has not already been authenticated (a session exists between the OpenID provider and the client if he has been authenticated). While the authentication scheme is not defined in OpenID, HTML forms are usually used. The OpenID provider redirects the client back to the relying party (Fig. 6. 2-3, 2-4); this redirecting URI includes Joe's identifier (shown as URI form http://joe... in Fig. 6), which is e-signed by the OpenID provider. If the electronic signature is successfully validated by the relying party, a session state is updated at the relying party (shown as cookie sessions in Fig. 6). No actual session management scheme is defined in OpenID, but usually cookies are employed (Fig. 6. 3-1, 3-2).

References

[1] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

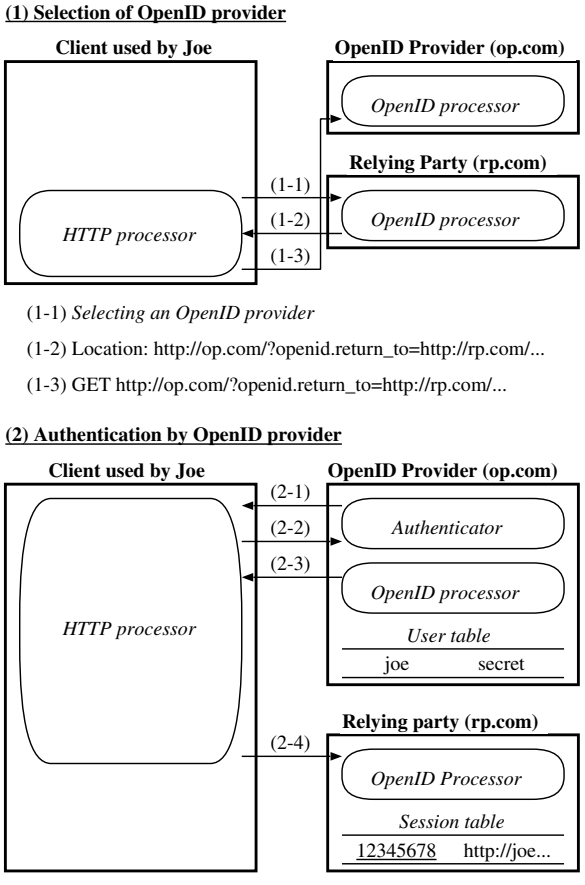

Fig. 6. An example of component interaction in OpenID; step (1) is followed by steps (2) and (3).

[2] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002.

[3] D. M. Kristol and L. Montulli, "HTTP state management mechanism," IETF RFC 2109, Feb. 1997.

[4] ——, "HTTP state management mechanism," IETF RFC 2965, Oct. 2000.

[5] D. Recordon and D. Reed, "OpenID 2.0: a platform for user-centric identity management," in *Proceedings of the second ACM workshop on Digital identity management*. New York, NY, USA: ACM, Nov. 2006, pp. 11–16.

[6] M. Atwood *et al.*, "OAuth core 1.0," OAuth Core Workgroup, Dec. 2007, http://oauth.net/core/1.0/.

[7] P. Wallace, *i-Mode Developer's Guide*, A. Hoffman, Z. Blut, K. Barrow, and D. Scuka, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., Apr. 2002.

[8] J. Franks, P. M. Hallam-Baker, J. L. Hostetler, S. D. Lawrence, P. J. Leach, A. Luotonen, and L. C. Stewart, "HTTP authentication: Basic and digest access authentication," IETF RFC 2617, Jun. 1999.

[9] T. Inoue, Y. Katayama, H. Sato, and N. Takahashi, "iAuth: HTTP authentication framework incorporated with HTML forms," in *Proceedings of the 5th IEEE International Symposium on Web and Mobile Information Services*, Apr. 2010.

[10] W. Eckerson, "Three Tier Client/Server Architecture: Achieving Scalability, Performance and Efficiency in Client Server Applications," *Open Information Systems*, vol. 10, no. 1, Jan. 1995.

[11] R. Kurebayashi, K. Obana, H. Uematsu, and O. Ishida, "A Web Access SHaping method to improve the performance of congested servers," in *Proceedings of 7th Asia-Pacific Symposium on Information and Telecommunication Technologies*, Apr. 2008, pp. 120–125.

[12] I. Zoratti, "Delivering Web 2.0 applications with MySQL and memcached," MySQL Conference, Oct. 2008.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Oct. 2007, pp. 205–220.

[14] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004.

[15] A. Lakshman, P. Malik, and K. Ranganathan, "Cassandra: A structured storage system on a P2P network," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Products Day #1*, Jun. 2008.

[16] A. Mesbah and A. van Deursen, "An architectural style for Ajax," in *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, Jan. 2007, p. 9.

[17] J. R. Erenkrantz, M. Gorlick, G. Suryanarayana, and R. N. Taylor, "From representations to computations: the evolution of Web architectures," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, Sep. 2007, pp. 255–264.

[18] R. Khare and R. N. Taylor, "Extending the representational state transfer (REST) architectural style for decentralized systems," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 428–437.

[19] D. Hatebur and M. Heisel, "Problem frames and architectures for security problems," *Lecture notes in computer science*, vol. 3688, pp. 390–404, Oct. 2005.

[20] P. Fenkam, H. Gall, M. Jazayeri, and C. Kruegel, "DPS: An architectural style for development of secure software," in *Proceedings of the International Conference on Infrastructure Security*. London, UK: Springer-Verlag, Jan. 2002, pp. 180–198.

[21] J. W. Yoder and J. Barcalow, "Application security," in *Proceedings of 4th Conference on Patterns Languages of Programs*, Sep. 1997, http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/.

[22] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A security architecture for computational grids," in *Proceedings of the 5th ACM conference on Computer and communications security*, 1998, pp. 83–92.

[23] D. Reed, M. Le Maitre, B. Barnhill, O. Davis, and F. Labalme, "The social Web: Creating an open social network with XDI," *PlaNetwork Journal*, Jul. 2004.

[24] The Liberty Alliance, http://www.projectliberty.org/.

[25] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web services: concepts, architectures and applications*. Springer Verlag, Sep. 2003.

[26] L. Richardson and S. Ruby, *RESTful Web services*. O'Reilly Media, Inc., May 2007.

[27] R. T. Fielding, "A little REST ard relaxation," ApacheCon Europe, Apr. 2008.