EOS: <u>E</u>xactly-<u>O</u>nce E-<u>S</u>ervice Middleware

German Shegalov * Gerhard Weikum * Roger Barga ** David Lomet **

* University of Saarland
D-66123 Saarbruecken
GERMANY
{shegalov, weikum}@cs.uni-sb.de

** Microsoft Research
Redmond, WA 98052
U.S.A.
{barga, lomet}@microsoft.com

1 Problem Statement

Today's web-based E-services do not handle system failures well. One of the most prominent examples is unintentional purchase of multiple copies of the same item (e.g., a DVD) in an online store. This may happen when the user sees a browser timeout for the final "checkout" ("place order") request caused by a short outage or overload of the network or the backend servers (typically during peak load*)*. Whereas the request may have been successfully albeit slowly processed, the user may attempt to send the check-out request once again, e.g., by hitting the browser "refresh" button, unintentionally buying another copy of the same item.

Another example is a home-banking application deployed by one of the biggest German banks. This application uses a so-called PIN/TAN security procedure. Each user is identified by a personal identification number (PIN). The bank hands over a list of transaction numbers (TANs) to each user. A TAN must be provided for each home-banking transaction to be accepted. For security reasons each TAN can be used only once. The following problem may arise (and has indeed happened to customers). After the first attempt to issue a money transfer order the user perceives a long delay resulting in an error message stating "this page is currently not available". The user re-submits the request and the "resurrected" application responds with: "A TAN was used twice. Your TAN list has been frozen. Please contact your nearest branch office if you would like to have your TANs reactivated again".

Such phenomena occur because the "stateless" interaction paradigm of the web puts the burden of managing sessions, and in particular handling failures, on application programs. Unfortunately, failure handling logic can be fairly complex, and application programs often make errors when responding to errors. In particular, they may simply forget actions already taken, not only after a successful execution but also after a system failure, so that they cannot guarantee exactly-once execution.

In contrast, our approach aims to place failure handling logic into a generic Internet middleware framework so that failures are masked from application programs (and users). Application programs are thus relieved from handling message timeouts and other exceptions caused by system failures. Based on the conceptual work in [2], we have developed a prototype system, coined EOS, that uses Microsoft's IE5 browser on the client side and the popular Apache/PHP middleware as the middle tier of three-tier Web applications. With our specific modifications to the IE5 environment and the PHP servlet engine, the EOS prototype guarantees exactly-once execution for all requests. Our modifications are transparent to the application programs: no changes are required to servlet programs (i.e., PHP scripts) and no failure handling code is required by these programs other than application-level exceptions such as "item out of stock" etc. and dealing with back end transaction aborts. As a result, all business requests, including those with non-idempotent effects, are processed such that their effects occur exactly once. This guarantee includes messages seen by applications and users as well as data updates issued to backend servers.

In addition to [2], conceptual work on recovery guarantees for Internet applications includes [3,5,7]. However, to our knowledge, our prototype is the first work that provides an implemented solution.

2 Framework

A framework for exactly-once guarantees in general multi-tier systems by means of interaction contracts (IC) between communicating components is described in [2]. There are three types of components considered in this framework: eXternal components (XCom's) modeling human users and parts of the system outside of the framework, Persistent (PCom's) representing mid-tier applications, and Transactional (TCom's) modeling ACID resource managers like database servers.

The framework requires a ***Committed Interaction Contract*** **(CIC)** between two PCom's (e.g., the client program and the mid-tier application program). This requires that the sender of a request or reply message prom-

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

ise that it can always recreate the message and also its own state as of the time of the message or more recent. In this way a PCOM can resend the message if demanded by the receiver; the receiver, on the other hand, must guarantee that it can detect and eliminate duplicate messages. The sender is released from its promise in a second step of the CIC protocol once the receiver can itself recreate the message if needed upon a failure.

Another type of contract is the ***External Interaction Contract*** **(XIC)** between an XCom, typically a human user, and a PCom, typically the client software of that user. In this setting the XCom does not promise message recreatability for user input, and it is not necessarily able to eliminate duplicate output. Consequently, not all failures can be perfectly masked in this setting. However, a PCom in an XIC, which includes recreatability of PCom-sent messages and PCom state, still provides automatic failure masking in most situations.

The central theorem in [2] proves that virtually all system failures (with some inescapable exceptions should a failure occur during an XIC, e.g. with a human user) can be masked, and each request is processed exactly once. Our prototype system aims at an efficient implementation of the framework outlined above in the internet context.

3 System Overview

Figure 1 depicts a typical E-Service architecture we have chosen for the prototype implementation. An E-Service in this setting encompasses a PHP engine connected through the ODBC driver manager to a data server.

An E-Service can be invoked either in a business-to-customer manner by a human user via her web browser (IE5 in our prototype) or in a business-to-business manner by some other E-Service. According to [2] we establish an eXternal IC between the end user and web browser, and a Committed IC (CIC) between PHP and the web browser. In addition, we plan to establish a CIC between PHP and the ODBC ***driver manager***, and finally a Transactional Interaction Contract (TIC) is between the ODBC driver manager and the data server, using techniques from [1]. Note that these techniques are complementary to the ones in our prototype; the work in [1] exclusively focused on ODBC sessions and did not address Internet middleware issues and the handling of client failures.

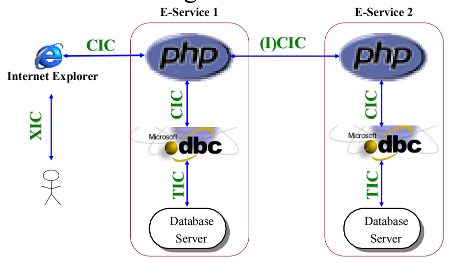

Figure 1: Typical E-Service Architecture

Either a CIC or Immediately Committed IC (ICIC) is used for interactions between a pair of E-Services. With ICIC, both E-Services are able to recover autonomously after a successful interaction, if needed (see [2]).

3.1 Browser Extensions

The recovery and logging routines for the web browser are implemented in JavaScript with extensive DHTML event processing, as supported by IE5. They are added to the application's HTML output by the web application server in the original response to the client. No explicit changes to the original HTML code are needed.

3.1.1 Client Logging

The main function of our event-handling code stubs is to implement an XIC between user and browser. It does this by logging the user updates to the browser state (e.g., the user having clicked a button or filled in a form entry). This logging is done by modifying a so-called XML store which is an XML structure managed by IE on the client's disk in a way similar to a persistent cookie. This feature is provided by IE with a default persistence behavior called "userData Behavior". HTML elements with attached userData behavior provide the methods for accessing the individual elements of the XML store. The XML object associated with the XML store can also be accessed and manipulated using the XML DOM parser which is natively supported by IE [4].

For the IE browser side of the CIC with the web application server, all relevant state information (i.e., each input field, whose value is passed to the web server, so that a completely identical http request can be generated or "replayed" again) and rendering details (the window area viewed or edited by the user) are logged to an XML store associated with the current session step. This is implemented by intercepting an "*onPropertyChange*" event. To force our log entries in the XML store to stable storage, we call the "*save*" method which triggers IE to write the XML store to disk.

3.1.2 Client Recovery

Client state survives failures as follows. When the browser fails and the user restarts it and revisits the same e-service initialization page, the browser is automatically redirected by the web server to the last visited (i.e., most recent) page of the interrupted conversation and our JavaScript code will be reloaded. The JavaScript code is set up to first look for an XML store previously saved on the client machine. If the XML store exists, its contents are used to recover input field values and replay all relevant events on windows, buttons, and forms, so that the user would not see any difference to the state immediately before the failure.

3.2 Application Server Modifications

To implement a CIC at the web application server, the main issues to be addressed are the virtualization of message ids and the logging of http requests and replies as well as session state information at the server side. To minimize our source code modifications to the regular infrastructure on the server, we refrained from making any changes to the web server or the Zend engine's kernel and only modified the PHP session management module.

For virtualization, we use our own message sequence numbers (MSNs) that are unique and consecutive within an application session. Such a session might consist of the stepwise filling of a shopping cart or various forms for tax declaration, steps that constitute a "logical (and stateful) session". (This notion of session is independent of the underlying TCP sessions.) These MSNs are added to the http request and reply messages in the form of an additional cookie. This is a natural and easy extension to the Zend engine, as it already encodes a form of session id in a cookie. Session ids in PHP applications are optional and activated either explicitly by the *session_start* function or implicitly by the first invocation of the *session_register* function. The kinds of e-service applications that we are interested in would typically use this feature. However, we do not depend on the original PHP application already using sessions; we always activate a session by setting *session.auto_start*=1 in the PHP configuration file.

3.2.1 Application Server Logging

On the browser side, we rely on IE5 to make cookies persistent; on the server side, the modified Zend engine invokes code to force-log all http messages, including cookies in message headers. When a client re-sends the same http request, it includes the original cookie with the original MSN, so that the application server (i.e., our code in the modified Zend engine) can easily detect duplicates. As the last used MSN is always carried by the cookie that is sent back and forth between client and application server, MSNs can be easily made consecutive even with "non-sticky" connections to a web server cluster. When cookies are disabled in the browser, we apply a similar technique using hidden form fields or URL-encoded parameters, which are the Zend engine's alternative methods for session ids. (Having cookies enabled has become a de-facto standard requirement for most E-service applications.)

In addition to encoding the MSN into the session-id cookie, the modified Zend engine adds JavaScript code to the HTML page that is returned by the PHP application program invoked via the http request. This serves to implement the client side of the CIC and also the XIC between the client and the user (see Section 3.1 above).

The modified Zend engine force-logs all outgoing http replies and tags each of them with the MSN included in its header. This is done by writing this information, in the form of additional session variables, to the PHP session state file. Note that there is no need to force-log the incoming http request, as the client already promises the persistence of this message by its part of the CIC. The session state file is accessible to all clones of PHP server processes that are controlled by the web server. If multiple web servers are set up in a computer cluster for the same IP address, the file must be shared among all nodes in the cluster. This technique ensures that we do not depend on "sticky" connections between http clients and PHP server processes.

For a server to proactively recover PHP servlet results after a server crash, as opposed to replaying servlets only when prompted by re-sent client requests, we can optionally log incoming http requests along with the PHP variables filled by http *get* or *post* parameters and all session variables that have been registered up to this point. The log record for an http request already contains the name of the invoked PHP program and its input parameters, which are either form variables or encoded in the URL. This, together with the persistently logged session variables, captures the initial state of the servlet execution. Since the servlet is piecewise deterministic (PWD), no other logging is needed for correctness.

3.2.2 Application Server Recovery

We can now describe how the modified Zend engine handles the various exceptions and recovery situations:

Upon receiving an http *get* or *post* request that carries a cookie with an MSN, the log is checked to determine if this is a duplicate request. If it is a resent request and the corresponding servlet has already terminated and produced an http reply, the reply is retrieved from the log and sent to the client. If the servlet has died and no reply is available, the servlet is restarted. If we had logged the state of session variables during the prior incarnation of the servlet, its replay would start from the last completed installation point.

When an http reply is sent to a seemingly dead client (e.g., the client does not acknowledge the message at the underlying TCP level), the server simply ignores this "problem" but is prepared to receive a duplicate of the client's original http request at a later point. When this client-initiated prompting happens, the server re-sends the reply. As the reply itself is stably logged at the server, it is guaranteed to persist across server failures. Once the client acknowledges the receipt of the http reply by issuing another http request within the same application session or invokes a servlet with a *session_destroy* function call, the server can garbage collect the previous step's log entry. To alleviate the potential danger that the server cannot safely discard log entries for clients with users who have intentionally aborted sessions or simply walked away, we can enhance the JavaScript code that we embed in the http replies to react to "*onAbort*" browser events and to timeouts: the code would simply send an explicit "abort session" message as a final http request to the web application server.

When a PHP process fails while executing a servlet on behalf of a client's http request, recovery is initiated when the client repeats its request (as part of the CIC behavior) or the user hits the "Refresh" button. This resent request will be handled by the next available PHP process, and that process will first perform a duplicate elimination test, possibly replay the servlet execution, and finally (re-)send the reply. When the web server or the entire computer fails, the same thing happens after the system restart. So recovery is automatic, but we rely on the client re-initiating the request rather than on server initiative. This behavior carries over to a web server farm on a computer cluster; failover to another node in the cluster is automatic as long as clients resend requests.

Logging incoming http requests is optional. When it is used, however, it is equivalent to logging http replies. Then reply logging would be optional. A request log record can thus be seen as a form of logical log entry that enables recreation of its associated reply, with a reply log record a form of physical log entry for the result of the servlet execution. Having both options provides us with optimization opportunities. Physical reply logging avoids the re-execution of servlets and thus provides faster recovery. On the other hand, there are applications that may create very large reply log entries, and it could be more efficient to re-create these replies by replaying the request. This holds, for example, for applications with large downloads such as image, video/audio, or software repositories.

3.3 Run-time Overhead

To evaluate the run-time overhead of our failure-masking techniques and exactly-once guarantees, we performed measurements with Apache/1.3.20 and the Zend engine (PHP/4.0.6) running on a PC with a 1 GHz Intel Pentium III and 256 MB memory under Windows2000. The load on this web application server was generated by a synthetic http request generator (Microsoft Web Application Stress Tool). The generator simulated conversations of *n* steps, each of which simply sent three string parameters as form fields, and a simple PHP program incremented a

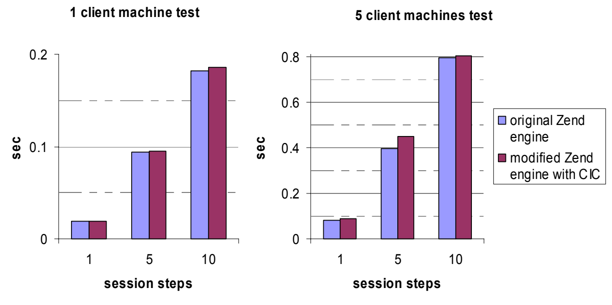

Figure 2: Overhead Measurements

counter registered as a session variable and returned its value to the client. There were no human user interactions or simulated think times in this experiment. The left chart in Figure 2 shows the total elapsed time, between the first request and the last reply as seen by the client, and the CPU time on the server side for *n = 1, 5, 10* steps, comparing the original Zend engine to the modified Zend engine with CIC behavior for exactly-once guarantee. The chart shows that the overhead of our CIC implementation is almost negligible, with respect to both user-perceived latency and increased CPU time.

We also performed multi-user measurements where the http request driver was replicated on 5 different client machines, each of which generated requests to the same web application server without simulating any think times (i.e., using a closed system model). The right chart in Figure 2 shows the measured average response time and throughput in terms of the simulated *n*-step user sessions. The performance degradation is less than 10 percent and thus well within the range of acceptable overhead.

4 Demo Description

For the demo, we use a simple PHP script, which opens a session upon the first HTTP *get* request. The state of this simple web application consists of a counter variable, the value of which is incremented by 10 in each session step (i.e., upon each subsequent script execution requested from the browser). The demo does not include interactions with data servers since this part is not yet included in the current prototype.

We show that the original PHP engine is not able to provide exactly-once execution. Note, that we do not blame PHP specifically, since all present web application technologies like ASP, JSP etc. are susceptible to the issues discussed in Section 1. Then we contrast this discouraging experience with the same experiment, which we carry out with the IE5 browser and the PHP engine, both enhanced in a way that they now transparently implement the appropriate interaction contracts. As a result, the same PHP script (without any changes in its source code) is now executed properly even in the presence of system failures on the server side and also when the web browser crashes. In addition, the external interaction contract for the web browser minimizes the user's need for having to retype input after a browser failure.

5 References

[1] Barga, R, D. Lomet, T. Baby, S. Agrawal: *Persistent Client-Server Database Sessions*; Int. Conference on Extending Database Technology (EDBT), Konstanz, Germany, 2000

[2] Barga, R., D. Lomet and G. Weikum: *Recovery Guarantees for General Multi-Tier Applications*, Int. Conference on Data Engineering (ICDE), San Jose, CA, 2002

[3] Dutta, K., D. VanderMeer, A. Datta, K. Ramamritham: *User Action Recovery in Internet SAGAs*, Int. Workshop on Technologies for E-Services, Rome, Italy, 2001.

[4] MSDN Library: *Web Development/ Behaviors/ Persistence/ ASP*, http://msdn.microsoft.com/

[5] Pedregal-Martin, C., K. Ramamritham: *Guaranteeing Recoverability in Electronic Commerce*, Int. Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, 2001.

[6] PHP, *Documentation and Downloads*, http://php.net/

[7] Schuldt, H., A. Popovici, H.-J. Schek: *Automatic Generation of Reliable E-Commerce Payment Processes*, Int. Conference on Web Information Systems Engineering, Hong Kong, 2000.

[8] Zend: *Documentation and Downloads*, http://zend.com/