EOS^2: Unstoppable Stateful PHP

German Shegalov*, Gerhard Weikum
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85,
66123 Saarbrücken, Germany
german.shegalov@acm.org, weikum@mpi-inf.mpg.de

1. INTRODUCTION

A growing number of businesses deliver mission-critical applications (stock trading, auctions, etc.) to their customers as Web Services. These applications comprise heterogeneous components distributed over multiple layers. They pose strong requirements for service and consistent data availability from both legal and business standpoints. Since many systems count many millions of lines of code, some bugs pass quality assurance undetected which leads to unpredictable service outages at some point.

Recovery in transactional systems guarantees: i) that an operation sequence declared as a transaction is executed **atomically** (either completely or not at all when interrupted by a failure) ii) and that completed transactions **persist** all further failures. Atomicity and persistence do not suffice to guarantee correctness. It is the application that needs to handle timeouts and other exceptions, retry failed requests to servers, handle message losses, and prepare itself with a full suite of failure-handling code.

Incorrect failure handling in applications often leads to incomplete or to unintentional **non-idempotent** request executions. This happens because many applications that are **stateful** by nature, i.e., with a state remembered between consecutive interactions, are rendered **stateless,** where all interactions are independent for easier manageability. Consequently, timeouts and resent messages among application servers or Web Services may lead to unintended duplication effects such as delivering two tickets for a single-ticket purchase request, resulting in severe customer irritation and business losses. The standard solution in the TP and DBMS world requires all state information of the application to be managed in the database or in transactional queues [4, 7, etc.], but this entails a specific programming discipline that is often viewed as an unnatural burden by application developers.

The interaction contracts (IC) framework [3] provides a generic solution by means of integrated data, process, and message recovery. It **masks** failures, and allows programmers to concentrate on the application logic, greatly simplifying and speeding up application development. A challenge in implementing this kind of comprehensive multi-tier application recovery is to minimize the overhead of synchronous disk writes **forced** by the recoverability criteria**.** While message and process recovery has a rich body of literature [6], IC's improve the prior state of the art in terms of efficiency and flexibility regarding autonomous component recovery. The need for new recovery techniques has also been raised by [5]

This paper presents a new version of the **E**xactly-**O**nce Web **S**ervice platform EOS^2 whose initial prototype has been described in [3, 9]. EOS^2 is a major advancement over EOS and differs significantly from its predecessor. EOS^2 masks failures from the end-user in a sense that she does not perceiver transient failures in the middle tiers and the backends whereas more severe failures require only that she revisits the greeting page, and her session is automatically restored without any data loss. EOS^2 masks failures from the DHTML/AJAX developer in a sense that she does not need to write the browser crash and the network outage handling code to achieve what is described above. EOS^2 masks failures from the PHP developer in a sense that she does not need to care about PHP crashes and network outages when calling remote PHP scripts. EOS^2 is able to replay arbitrarily structured *n*-tier PHP applications with interleaved accesses to shared data, whereas EOS was limited to two-tier applications with a frontend browser and a single PHP backend node accessed without race conditions. The improvements to the original PHP Session module include: I/O-efficient log structured access of session data, LRU buffers for state and log data, and improved concurrency through spinlock-based shared/exclusive latches.

1.1 Review of Interaction Contracts

The framework considers a set of interacting components of three different types. (i) **Persistent** components (Pcom), (clients, application servers) are able to recreate their state and messages as of the time of the last interaction upon crashes, and eliminate duplicates; (ii) **Transactional** components (Tcom), e.g., database servers, provide thesame guarantees only for the final interaction (commit request/reply); (iii) **eXternal** components (Xcom) without any recovery guarantees represent human users and legacy components that do not comply with the framework.

The Pcom is a central concept of the framework. Pcom's are **piecewise deterministic**, i.e., their execution is deterministic up to nondeterministic input from other components. A Pcom's state as of any particular time can be recreated via **deterministic replay** of the recovery log where the nondeterminism of the original execution is captured.

The exactly-once execution in the overall system is guaranteed when components obey to the obligations of the IC's. Interactions between two Pcom's must comply with either the **Committed IC** (CIC) or the **Immediately Committed IC** (ICIC). Pcom's and Tcom's exchange messages using the **Transactional IC** (TIC). Xcom's communicate only with Pcom's as determined by the **External IC** (XIC). CIC and ICIC are defined below. Please see our previous publications for details about TIC and XIC [3, 9].

1.1.1 Commited IC and Imediately Committed IC

The sender part of the CIC consists of the following obligations. The sender promises: **(s1)** that its state as of the time of the message send or later is persistent; **(s2a)** to send the message repeatedly (driven by timeouts) until receiver releases it (perhaps im-

*Current affiliation: Oracle, Portland Development Center, OR, USA.

VLDB '06, September 12–15, 2006, Seoul, Korea.

```
01.  <?php
02.  session_start();
03.  if(isset($HTTP_SESSION_VARS["count"])==FALSE)
04.    $HTTP_SESSION_VARS["count"] = 0;
05.  $HTTP_SESSION_VARS["count"]++;
06.  echo "Hi, I have been called ";
07.  echo $HTTP_SESSION_VARS["count"];
08.  echo "times\n";
09.  ?>
```

Figure 1: Sample Usage of PHP Session Support

plicitly) from this obligation; **(s2b)** to resend the message upon explicit receiver request until the receiver releases it from this obligation; **(s3)** that its messages have unique identifiers, typically **message sequence numbers (MSN)**.

The receiver promises: **(r1)** to eliminate duplicate messages (which sender may send to satisfy s2a); **(r2a)** that before releasing sender obligation S2a, its state as of the time of message receive or later is persistent without periodical resend by the sender; **(r2b)** that before releasing the sender from obligation S2b, its state as of the time of the message receive or later is persistent without the sender assistance. After s2a release, the receiver must explicitly request the message from the sender; the interaction is **stable**, i.e., it persists (via recovery if needed) over subsequent crashes with the same state transition as originally. After s2b release, the interaction is **installed**, i.e., the sender and the receiver can both recover autonomously.

The **Immediately Committed IC** is a CIC where the receiver immediately installs the interaction (usually by adding the complete message to the stable log), such that the sender is released from the obligation s2a without entering the obligation s2b.

1.2 Technology Background

1.2.1 PHP and the Zend Engine

PHP (recursive acronym for "PHP: Hypertext Preprocessor") [8] is a widely used general-purpose scripting (i.e., interpreted) language that is especially geared for Web application development. PHP is platform-independent and is available as an add-on module for a wide variety of Web servers. PHP is used with more than 50% of the Apache Web Server installations [1, 2]. The PHP implementation is an open-source project consisting of many **PHP modules** responsible for different function subsets of the PHP language and the Zend engine implementing the language interpreter [11]. Most of PHP's syntax is borrowed from *C* with a small fraction of elements coming from *C++*, *Java*, and *Perl*. PHP owes its popularity to the ease of its syntax and its expressive power. In the rest of the paper, the term PHP is used to refer to PHP 4.0.6 that served as the basis of our prototype.

1.2.2 PHP Session Management

PHP is shipped with a **session module** for maintaining the **PHP application state** across multiple HTTP requests. The session module supports various methods of storing the session state (e.g., in the file system, shared memory, central database, etc.). The state of a PHP application may be private (e.g., a shopping cart) or shared, concurrently accessed by multiple users (e.g., the highest bid in an electronic auction). The state variables, accessed by their string names through the global session array *$HTTP_SESSION_VARS*, may be of any basic or derived data types except resources. PHP typically uses a cookie to propagate the session (state) id to the Web browser.

The session support is activated either explicitly by calling the function *session_start* or it is started automatically, if configured

```
01.  <?php
02.  $b2b=curl_init("http://eosauctions.com/b2b/");
03.  $par=array("auct_id"=>100232, "bid"=>50.74);
04.  curl_setopt($b2b, CURLOPT_POST, TRUE);
05.  curl_setopt($b2b, CURLOPT_POSTFIELDS, $par);
06.  $b2b_reply=curl_exec($b2b);
07.  curl_close($b2b);
08.  ?>
```

Figure 2: Sample Usage of the CURL Module

appropriately, prior to executing the PHP code for the given request. The session module reads the state associated with the session id provided with the request from the session storage and makes it accessible for the PHP script as the session array. The new state is made available for subsequent requests by either calling the function *session_write_close* or implicitly when the script terminates.

The script of Figure 1 counts how frequently it has been invoked. It looks up the current state (line 2), zeroes the state variable *count* upon the first access (lines 3-4), and increments this variable for each invocation (line 5). If a script maintains a shared state for several users, their accesses must be serialized for consistency. The session module relies on the **session storage module** in current use to achieve this. The standard session storage module that makes use of the server's file system exploits file locking for this purpose. This, however, works only on UNIX systems where requests are executed by different server processes. Our prototype provides a more flexible concurrency control mechanism coined *latches* based on *spinlocks*.

The standard session storage module allocates a new file for each session upon the very first call to the function *session_start* that exists until the function *session_destroy* is called at the end of the session. The original implementation does not have a cache in the main memory, and performs a random I/O for every access to a particular PHP application state. We improve this by implementing LRU caching.

1.2.3 PHP Business-to-Business

PHP offers several options to interact with (potentially PHP-enabled) Web services. One of the most popular and elegant methods provides the CURL module that allows PHP applications to access remote HTTP and many other resources in a uniform fashion. The complexity of the protocols is hidden behind a simple interface that keeps the coding effort at minimum. This functionality is implemented by the CURL library for *C* [10].

Figure 2 shows a fragment of a PHP script that uses CURL to call another Web service to bid for an auction item. It first initializes a resource variable for this operation (line 2) and defines its parameters in an array variable (line 3). Next, the PHP script specifies that the POST method should be used, and associates the parameter array with the request (lines 4-5). Some Web Services are invoked via SOAP, the Simple Object Access Protocol layered on top of HTTP. When SOAP is involved, we would pass a SOAP message as a POST parameter. The HTTP request is sent to the URI provided during the resource initialization and the reply string is assigned to the variable *$b2b_reply* (line 6). When the CURL resource is no longer needed, it can be either explicitly closed (line 7) or it is garbage collected automatically at some point after the termination of the script.

Note that by implementing a CIC for CURL and Session modules of PHP, recovery guarantees are provided at the HTTP layer and thus, higher-level applications including PHP script libraries

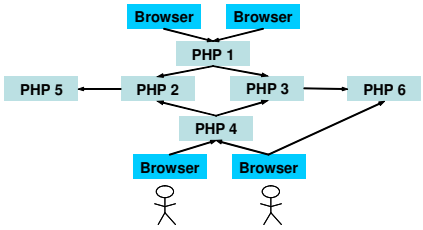

Figure 3: Sample EOS^2 Web Application

for SOAP and other protocols over HTTP are relieved from dealing with system errors at all.

1.3 Contribution

Using the technologies and the components described in the previous section developers are able to build arbitrarily distributed (potentially stateful) Web applications. The EOS^2 software enhances all components in this distributed environment with interaction contracts and their strong recovery guarantees. This masks all transient failures like component crashes or message losses to the application, thus greatly relieving the programmer from writing exception handling code. EOS^2 server part is implemented on multithreaded Apache 1.3.20 within PHP 4.0.6 for Windows XP. Our recovery layer comprises less than 10,000 lines of *C* code including modifications to the Zend engine, its Apache API, and the PHP session module.

Figure 3 sketches a sample Web application. End-users call Web applications on the server sites PHP 1 and PHP 4 using their browsers. PHP 1 invokes (through the CURL module) Web Services on PHP 2 and PHP 3 that in turn call PHP 5 and PHP 6, accordingly. The frontend Web application server PHP 4 usually behaves identically; occasionally, however, it optimizes this execution path by having the browser immediately invoke PHP 5 from one of the embedded HTML elements. EOS^2 enables failure resilience and masking in the entire system. To comply with the IC framework, the PHP application servers and the browsers are turned into Pcoms whereas the end-users can only be viewed as Xcoms. The interactions between a browser and an end-user must follow the XIC. PHP servers interact under the CIC or the ICIC. Note that the contracts are established and enforced automatically by EOS^2 and are completely transparent to the Web application.

2. PERSISTENT BROWSER

There are two major design goals of EOS enhancements to the browser. The first is to improve the user experience by saving as much of her input across a failure (a browser crash or Web Service unavailability) as possible. This avoids the need for annoying repetition of long inputs, which may happen with lengthy forms such as e-government applications (tax declarations, visa applications, etc.) and e-business applications (e.g., insurance and credit approval requests). This is the task of the XIC implementation for the browser. The second design goal is to give the guarantee to the end-user that all server requests are executed exactly once, which is the task of the CIC stub of the browser.

Browser recovery is implemented using JavaScript and it has not changed since the initial prototype version described in [3, 9]. The server adds the browser logging and recovery code as the last step of output processing. Original server scripts do not have to be changed. Logging is done by modifying a so-called *XML store*, an XML structure managed by IE on the client's disk similarly to persistent cookies. Since we had no access to the browser source code, we require the user to revisit the greeting page (e.g., http://servername/) of the Web Service manually after a crash, in order to have her session restored automatically.

3. PERSISTENT PHP (EOS-PHP)

EOS-PHP is the major (middleware) part of our prototype. It can serve as both an HTTP server and a middle-tier HTTP client at the same time. It transparently implements the (I)CIC stubs for incoming and outgoing HTTP interactions with other PHP applications and Web browsers. EOS-PHP is geared to provide the recovery guarantees for stateful PHP applications. The log is provided as a universal storage for **log entries** identified by **log sequence numbers (LSN)**, and the **session state data** identified by a pair (LSN, session id). LRU buffers are used to save IO's. EOS-PHP controls concurrency using shared and exclusive latches.

EOS-PHP extensions are implemented in the *C* language. They encompass ca. 5500 lines of source code. For the implementation we used as much of the existing efficient Zend engine infrastructure as possible. Web server code is not changed.

When considering a single PHP Zend engine, we can distinguish three relevant system layers from the logging perspective. We observe HTTP requests at the highest level **L2**, individual PHP language statements at the middle level **L1**, and finally I/O calls to external resources such as the file system and TCP sockets (level **L0**). EOS-PHP does not support interactions with the file system, i.e., the PHP file system functions. Instead, EOS-PHP efficiently manages persistent application states stored as session variables. EOS-PHP does not deal with the PHP socket interface. Instead, EOS-PHP supports recoverable HTTP interactions through the CURL module.

A request execution by EOS-PHP breaks down into the following stages: client identification (Stage 1), URI recovery (Stage 2 for browsers only), reply resend (Stage 3), request execution (Stage 4), output processing (Stage 5). Note that Stages 2 and 3 are EOS-PHP operations needed for browser recovery. Prior to the request execution, a shared *activity latch* is obtained for the duration of the request execution. It prevents the garbage collection mechanism that uses this latch in the exclusive mode from physical reorganization of the log file.

3.1 Stage 1: Client Identification.

During request startup, EOS-PHP identifies the client id information submitted as cookies. If this information is missing, the client is assigned a new id and is redirected to the first session URI (browsers only). A B2B component (i.e., another EOS-PHP node) autonomously generates its id by concatenating its host name and TCP *listen port* (*socket*) number.

The following Stages 2, 3, and the state initialization part of Stage 4 are initiated on behalf of the function *session_start*.

3.2 Stage 2: URI Recovery.

Browsers need an additional stage for assisting in recovering the last message sent to the EOS-PHP engine. EOS-PHP checks if the current URI coincides with the URI that started the session (i.e., the greeting page URI). If so, we know that this is an browser revisiting the greeting page to restore the interrupted session. An empty page containing solely client recovery code is sent to back to the browser without incrementing the MSN cookie.

3.3 Stage 3: Reply Message Resend.

The log is consulted through the request message id lookup in the volatile **input message lookup table (IMLT) (client id, MSN, reply LSN)**, in order to determine if the HTTP reply is already

present. In the positive case, the HTTP reply is served right away and the current request is terminated. When the IMLT contains an entry for the request with the reply LSN being invalid, EOS-PHP is dealing with a request message resend: this thread is paused until the reply LSN is set, and the reply can be served. When the current request is not a duplicate, it is not terminated by this stage.

3.4 Stage 4: Request Execution

The request starts with fetching the PHP application state through the state buffer. If the entry for the current PHP application state could not be found (a new PHP session), the request inserts an empty state into the state buffer. A new LSN is generated for the request and EOS-PHP adds an initial log entry to the log buffer that contains PHP representation of the HTTP request and the translated PHP script file path. (additional variables can be marked for logging in the PHP configuration file if needed). An entry is also added to the IMLT containing the client id, the MSN of the message (both as submitted by the client cookies), and an invalid LSN. At this point the request thread latches the PHP application state in the shared or exclusive mode (as specified in the EOS^2-enhanced PHP function *session_start* that now accepts a Boolean flag *$read_only* as an optional argument). In contrast to the original PHP implementation, the ability to access the application state in the shared mode is an appropriate response to the fact that the load of e-commerce sites is dominated by read-only catalog browsing requests. The latch for the application state is held until the script calls the function *session_close* that replaces the original PHP function *session_write_close* to avoid irritation. If the request is declared as a *write* by calling *session_start*(*false*), the application state is stamped with the request LSN, whereas the volatile read LSN field of the buffer cell is updated in any event.

Nondeterministic functions treated by EOS-PHP include system clock reads (e.g., *time()* returning the current time, random value generators such as *rand(min, max)* generating a random number in the interval between *min* and *max*, and last but not least *curl_exec* returning output of a different Web Service. The routines asking for the system clock and random values are not only interesting because of their potential direct usage in a PHP script, but also because they are used to generate PHP session ids that are pairwise distinct with a high probability. This avoids a potential bottleneck of having a single node in a Web farm assign sequence numbers as session ids to all clients. The EOS^2-enchanced function *curl_exec*(*$handle*) implements the CIC transparently to PHP developers. One part of it the original code reporting failures to the user is replaced by a loop repeating requests on timeouts until the underlying *libcurl* function *curl_easy_perform* returns the success return code *CURL_OK*.

3.5 Stage 5: HTTP Output Processing

When the execution of the request is finished, EOS-PHP updates the reply LSN field of the request entry in the IMLT. In the current prototype solution, the log entries with HTTP output do not require immediate log forcing, since these messages are recreated during deterministic replay. In fact, for *curl_exec* requests EOS-PHP does not even create a log entry with the content of the outgoing message, just the reply is logged to resolve recovery dependency, as you saw above. The point is that EOS-PHP is able to send out the HTTP reply messages prior to forcing them to stable log. Therefore, EOS-PHP can lazily force output messages (from several KB to several MB) that are orders of magnitude larger than preceding log entries of the same request whose sizes range from less than 256 bytes to some KB. In addition, the browser

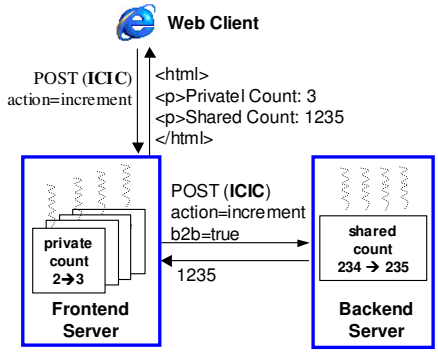

Figure 4: EOS^2 Demo Application

recovery code permanently cached in the main memory is inserted into original HTTP replies to browsers. This does not have to be logged of course, which otherwise would further increase the size of the output message log entry.

4. ABOUT THE DEMO

The call structure of the demonstrated application is shown in Figure 4. Both servers deploy the same PHP script. The script calls the nondeterministic function *time()*, reads the current application state, and increments the state variable *count*. On the backend server (receiving the flag *b2b* as a parameter), the accessed state is *shared* among all clients as specified by the explicit call to the function *session_id*. The new value of the state variable *count* is the only content of the HTTP reply body produced by the backend server. In contrast, the frontend server accesses a *private* state to increment the variable *count* and invokes another instance of this script on the backend server. The frontend server replies with a complete HTML page containing the shared and private *count* values to the browser.

In the demo, we invoke the Web application with two concurrent clients generating a random interleaving of PHP statements on the backend server. An arbitrary number of failures (backend and frontend server crashes, link outages) are injected. We show that our recovery replays all components in an exactly-once manner to the state intended by the original execution without failures. Additionally, browser recovery as in [3] makes the recovery guarantees complete.

5. REFERENCES

[1] *Apache HTTP Server Project*, http://httpd.apache.org/

[2] *Apache Module Report*, http://www.securityspace.com/

[3] Barga, R., D. Lomet, G. Shegalov and G. Weikum: *Recovery Guarantees for Internet Applications*, ACM Transactions on Internet Technologies (TOIT) 4(3), 2004

[4] BEA Tuxedo, http://bea.com/

[5] Candea, G. et al:.: *Microreboot -- A Technique for Cheap Recovery*, 6th Symposium on Operating System Design and Implementation (OSDI), Dec 2004, San Francisco, CA, USA

[6] Elnozahy, E., L. Alvisi, Y.-M. Wang, and D. Johnson: *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*. ACM Comput. Surv. 34(3), 2002

[7] *Oracle Advanced Queuing*, http://oracle.com/

[8] *PHP: Hypertext Preprocessor*, http//www.php.net/

[9] Shegalov, G., G. Weikum, R. Barga,, D. Lomet: *EOS: Exactly-Once E-Service Middleware*, 28th Int'l Conference on Very Large Databases (VLDB), Hong Kong, China, 2002

[10] Stenberg, D.: *cURL and libcurl*, http://curl.haxx.se/

[11] Zend Technologies, Inc. The PHP Company, http://zend.com/