TRANSFORM: A TRANSACTION SAFE WEB APPLICATION MODEL

Matthias Ihle and Georg Lausen
Institut für Informatik
Universität Freiburg
Georges-Köhler-Allee, 79110 Freiburg i. Brsg.
Germany

Keywords: Web applications, web forms, transactions, concurrency control.

Abstract: Classical models of database-driven web applications follow thin-client architectures, i.e., all data processing and business logic resides on the server while the client's role is to arrange and display the user interface. When transactions come into play, which may naturally span several consecutive interactions between client pages and server actions, problems arise since transactions cannot exceed the boundaries of server actions. We address this issue by proposing a novel architecture for web applications, where a webservice-based data access component is integrated directly in the markup of a client page. Our approach guarantees ACID transaction properties and generates serializable histories in the sense of conflict serializability. In contrast to past efforts to transaction management in web applications, our architecture does not necessitate the existence of an additional, external transaction server.

1 INTRODUCTION

Enterprise applications which are generally data-centric and transaction-based are often form-based systems with a submit/response style interface (Draheim and Weber, 2004). A proven architecture for such interfaces are thin client web applications, where the browser forms the interface tier and the application logic is implemented in different actions on the web server. These server actions deliver pages to the client and, eventually handle a database connection.

The human-computer-interaction works via the exchange of messages between client pages and server actions. In a typical scenario, some initial values are read from a database during a server action and displayed in web forms of a corresponding client page. Now a user can edit the data in the forms and send it to another server action that writes the changed values back to the database and displays the result in another client page.

Such a web form lifecycle, intended to happen in a single transaction, spans several transitions between client pages and server actions. Unfortunately it is not possible to embed the whole form processing into one single transaction of the underlying database system, since such transactions cannot exceed the boundaries of a server action.

The reason for this is that each server action needs a new database connection, since it is unpredictable in which process, respectively thread on the web server it is executed. Even persistent connections that provide a pool of open database connections cannot grant the assignment of the same connection in two consecutive server actions.

Usually, the problem is resolved by adoption of external transaction servers or an application specific solution.

The contribution of this paper is a novel model for web applications that provides a web service based database interface. Thereby we utilize the AJAX (Garrett, 2005) technologies of current browsers and access this interface via HTTP (Fielding et al., 1998) requests from within javascript. The access to this interface is defined directly in the markup of a web page.

The benefit of our model is that we can bundle all data access operations within a client page into a transaction satisfying the ACID properties (Härder and Reuter, 1983). For that, we do not even need additional components beside the web server.

In our model a transactional web page is created by a server action independently of the database state. Hence, a transaction in a client page is completely self-contained and there is no need to consider the

Ihle M. and LausenLausen G. (2006).
TRANSFORM: A TRANSACTION SAFE WEB APPLICATION MODEL.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Internet Technology / Web Interface and Applications*, pages 164-171
DOI: 10.5220/0001253101640171

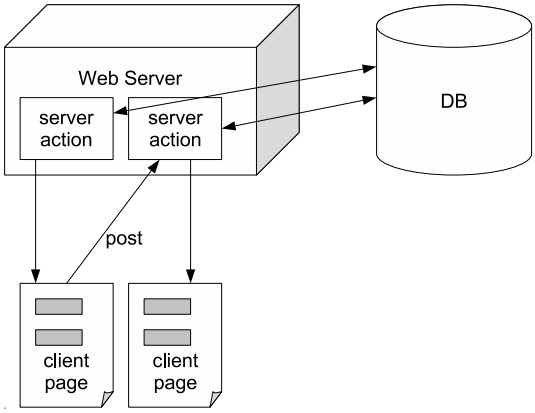

Figure 1: The classical form processing mechanism.

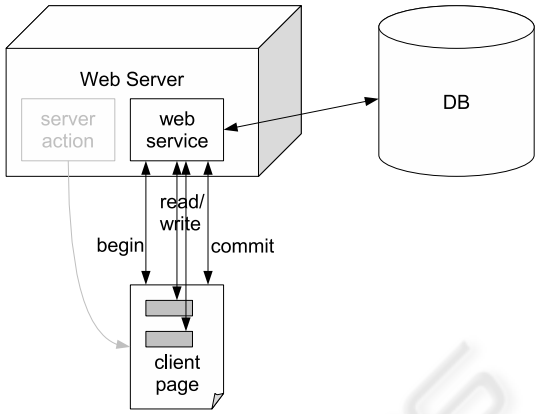

Figure 2: The processing of forms in TransForm.

preceding server action.

As it is typical, we assume that all data access happens via our interface. Otherwise the transaction manager of the database system interferes with that of our web service.

The remainder of this paper is organized as follows: section 2 presents the different parts of the TransForm model in detail, while section 3 describes the scheduler we have implemented in our prototype. Section 4 summarizes the field of research and finally, section 5 concludes the paper.

2 TRANSFORM

In TransForm, the data access during the lifecycle of a web page is enclosed in a transaction with the well known ACID properties. Actions of such a transaction, for instance, read initial values of form fields or write them back to the database, when edited. Additionally, there are actions that begin, abort or commit a transaction.

Therefore we do not use the standard form processing mechanism, depicted in figure 1. Here, several server actions and client pages are needed to process a form. A first server action creates a client page containing form values read from the database. Then, after changing the values, a user submits the form to a second server action which writes the new values back to the database and displays the result in another client page.

On the other hand, the TransForm procedure is shown in figure 2. The complete processing of the form is done within a single client page. The corresponding transaction is not influenced by the server action delivering the page, since the page generation happens independently of the database state. Therefore, in the figure this server action is depicted in grey.

We can simply assume a static web page with empty form fields containing definitions of the database objects that the form fields are associated with. After the page is loaded in the browser, a javascript program starts the transaction and reads the initial form values from the specified database objects. With the help of javascript event handlers this program now propagates user actions to the database, whenever a form field changes or the commit, respectively abort of the transaction is requested.

All these transaction operations happen via a stateless web service that constitutes the interface to the database in our TransForm model. Beside the requested operation a transaction identifier must be specified in every service invocation. This identifier is generated by the operation starting the transaction and hold by the javascript program in the client page. The web service now wraps all operations with the same identifier into a single transaction for which the ACID properties are guaranteed. Now, if the access to the database happens exclusively with this web service interface, TransForm produces conflict serializable schedules only. If other database interfaces are used simultaneously, the transaction manager of the database interferes with that of the web service. This way non serializable schedules may occur.

Behind the web service interface different scheduler strategies can be deployed. In our implementation we have opted for an optimistic strategy that is considered in detail in section 3.

All the information to access the web service, e.g. the connection of a form element to a database field or the URL of the service, in our model needs to be defined within a client page. Therefore, TransForm defines custom tags that are interleaved with the HTML tags in the markup of the web page. Every tag contains the database object it is connected with and stands for a particular HTML representation in the web page.

The benefit of this tag based data access facility is that it can be easily embedded into web application modelling frameworks, like (Ceri et al., 2000), that generate HTML pages in their output, since a TransForm page basically is an ordinary web page connected with an additional javascript program.

We will see in the remainder of this section how this works in detail. Therefore, we first present the data and transaction model before we describe the web service and the client interface in the following subsections.

2.1 Data Access Model

TransForm establishes an additional layer on top of the database system implementing its own data manager and transaction manager. The data manager addresses objects in the database and identifies conflicts among them.

Therefore it represents the actual database in XML. Now, objects in the database are addressed by XPath expressions (W3C, 1999). Although this additional layer certainly has its impact on performance, there are some incitements to do so:

- If we connect a web form element with a relational database field with the help of an XPath expression, the respective SQL statement can be automatically generated for read and write operations. Otherwise, we would have to state separate select and update statements for every form element.
- With this abstraction layer we can support not only relational backends but also native XML or document databases. We can even use storage facilities that come without native transaction support like regular file systems.
- Performance is in the majority of cases not a big issue because the system is mainly idle waiting for user inputs.

With the help of XPath expressions we now detect conflicts among the database objects, even when they occur because referential integrity constraints are defined among the tables in an underlying relational database. However, it would go beyond the scope of this paper to describe this functionality in detail.

Hence, for the remainder of this paper we simply assume a page model for the transactions where database object, specified by XPath expressions, are read and written.

Here, a data access operation simply is of the form $r(x)$ or $w(x, v)$. Thereby x is an XPath expression on the XML representation of the database specifying a database object. Furthermore, v is a value of the type of x. So, $r(x)$ reads and returns the value of x while $w(x, v)$ writes v to x.

The write operation covers insert, delete and update in the following way: an empty v means a delete of x while a non empty v is an update and an x that is not in the instance graph is interpreted as an insert.

2.2 Transaction Model

Besides the data manager, TransForm provides its own transaction manager that wraps single operations, called actions, into transactions.

In this model, an *action* is a pair (t, a). Thereby, t is a transaction identifier and a is either $r(x)$ or $w(x, v)$, the data access operations of the previous subsection, or one of the following transaction control operations:

- *begin*: starts a new transaction.
- *abort*: aborts the transaction.
- *commit*: tries to commit the transaction.

Now, a *transaction* t is an ordered sequence of *actions* that all share the same transaction identifier. Every transaction has exactly one *begin* action as its first step and either *abort* or *commit* as its last step.

2.3 Web Service Architecture

As we have defined transactions and their enclosed actions in the previous subsection we now need to adapt them in the web service. In essence, an action corresponds to an invocation of the web service.

The transaction identifier is thereby not an inherent part of the service, it is rather specified as argument by the service caller. So, a transaction is basically a subsumption of service invocations with the same transaction identifier.

With every service invocation, operations on the underlying database system are carried out. These operations depend on the actual protocol used by the scheduler. Thereby, it is important that the operations of a service invocation occur in a serial, non interleaved order with respect to the underlying database system. Additionally, the service call should happen in an atomic fashion. The simplest way to achieve this is to embed all steps performed during a service call in a native transaction of the database system. This is possible, because all the steps are executed during a single server action. If the database system does not have its own transaction facilities, some kind of locking mechanism needs to be provided.

Because we utilize the AJAX technology of current browsers to invoke the web service via HTTP requests in javascript, the use of the SOAP protocol (W3C, 2000) is inappropriate. We rather follow the REST (Fielding, 2000) web service design which has the following characteristics:

- The service is comprised of a resource which is named using a URL.
- The resource is accessed through a generic interface, in this case HTTP via the GET method.

- It is stateless, i.e. each request from the browser to service must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server. For instance, the identifier of the transaction a service invocation belongs to must be contained in a call argument.
- The response is an XML document. In our case, it contains information about success of operations, requested values or detected conflicts that lead to transaction aborts.

So the service is basically a HTTP call to the service URI where the requested action is specified in the GET variable `action`. Depending on the concrete action, further arguments are required. For the allowed actions this is as follows:

- *begin*: No further argument is needed. This action returns the transaction id.
- *abort*: The transaction id must be specified in the additional argument `tid`.
- *commit*: Here also the `tid` must be specified. The return value is either success or failure.
- *read*: In addition to the `tid` the `object` to be read needs to be given. The value of this object is returned.
- *write*: It takes the same arguments as *read* plus the value of the object via the `value` argument.

Example The following service invocation requests the commit of the transaction with the id 3. The service is located at the host `serviceurl` under the path `servicepath`.

```
http://serviceurl/servicepath
?action=commit&tid=3
```

The respective response to the above request could be the following XML document:

```
<response action="commit" tid="3">
    failed
    <conflict>563</conflict>
</response>
```

The response signals that the commit has failed due to a conflict with another transaction.

2.4 Browser Tags

Having the web service available, we need to embed it in web sites. Therefore, we introduce special tags in the page markup that encode all the necessary information like the database objects they are connected with in their attributes. These tags are converted into suitable HTML elements by a javascript program that is invoked when the browser has loaded the page. Together with normal HTML these tags build the skeletal structure of TransForm web pages.

The javascript program is additionally responsible for the following tasks:

- It first parses the document to identify all TransForm tags. Therefore it uses the DOM representation of the web page, an interface with whom the various page components can be accessed and manipulated.
- It starts a transaction for every *tf:form* element that it finds during the initial parse with a *begin* request. In the response to this request it gets an identifier for every transaction. Because this id is needed in all further service requests it is stored during the lifetime of the transaction.
- It takes care of the complete processing of the service requests. Therefore, event handlers are set accordingly and AJAX objects are created when such an event triggers. Furthermore, it handles the service response messages. Retrieved values are written to form elements and, in case of conflicts, transaction aborts are triggered.

In the following we present some selected tags that are used in the example in figure 3:

`<tf:form serv="url">` This tag forms the boundaries of a transaction. All further TransForm tags inside it belong to the same transaction. Additionally, it specifies the URL of the web service in the `serv` argument. With this mechanism we can define access to several databases in a single web page. This leads to distributed transactions that are explained in section 3.2.

This is similar to the ordinary HTML form element that specifies a server action and contains all the elements of the form.

`<tf:input xpath="x">` This tag is converted into an ordinary HTML input form element. It retrieves its initial value by a read request for object *x*. Additionally, an event handler is set that requests a write operation whenever a user changes the value in the form.

`<tf:abort>` Here, the javascript program creates a button with whom the user can explicitly request the abort of the current transaction. In such a case, a new transaction is started and the values of the form fields are retrieved once again from the database.

`<tf:commit>` This tag is also converted into a HTML button with whom the user can request the commit of the current transaction. Every *tf:form* tag must contain a commit button, otherwise the user can never request the commit of the transaction.

Figure 3 shows an example of a TransForm page. It is an ordinary HTML page with some custom tags that a browser does not display when rendering the page. In the page header a javascript file is included that

```
<html>
<head>
        <script src="transform.js">
</head>
<body onload="transform()" onunload="abort()">
        <div>
                <tf:form service="url" protocol="opt">
                        <tf:input xpath="a[b=2]/a">
                        <tf:input xpath="a[b=2]/c">
                        <tf:commit/>
                </tf:form>
        </div>
</body>
</html>
```

Figure 3: An example of an HTML document interleaved with TransForm tags.

contains the aforementioned program. The body element defines an *onload* event handler that invokes the *transform()* function. This function now converts the TransForm tags according to the above description and starts a transaction for the contained *tf:form* element. It sets the event handler for the *tf:input* and the *tf:commit* tags accordingly, for instance, an *onclick* event for the submit button.

For every event an appropriate handler is defined in the script. This handler creates an AJAX object for every request that asynchronously invokes the web service. Therefore, a callback function is defined that handles the response of the request.

Example scenario Suppose two users, X and Y, each load the above example page and edit the form fields depicted by the *tf:input* tags. In each browser a transaction is started and the initial values are retrieved via respective *begin* and *read* service calls. Let user X change the values of both input forms while user Y simply changes the value of the object $a[b = 2]/a$.

A scheduler would now detect a conflict because both transactions write the same object ($a[b = 2]/a$). Depending on the order of the commits requested by pressing the commit button and the particular scheduler strategy one transaction is accepted and committed into the underlying database while the other one is rejected and all its write operations are discarded.

At the end of this section it is clear that a TransForm page only makes use of standard web technology and therefore still is an ordinary HTML page with some non-standard tags and a dedicated javascript program. This has the benefit that it integrates well into web application frameworks that normally produce web pages in their output. Thereby, it provides them with a transaction safe data access component. It must solely be taken care that the generation of the web pages still is independent of the database that is used within the page. Otherwise the transaction safety cannot be guaranteed anymore.

3 SCHEDULER

Behind the interface presented in the previous section different scheduler strategies are possible.

Thereby, it is not necessarily clear that the same strategies proven for classical databases are also well suited in the context of the web environment.

Here we deal with online transactions where it is not possible to simply rerun a failed transaction. In such a case, the user needs to specify the values to be written in the database again depending on different read values.

Hence, we first gather the requirements of schedulers in the context of web applications and discuss the pros and cons of several classical schedulers strategies found in (Weikum and Vossen, 2001):

- The scheduler protocol must not constrict the form editing when several users work with the same database objects. It should only interfere when the current transaction cannot be committed anymore.
- If transactions are in conflict with each other the accepted transaction should be chosen after a 'first come first serve' principle.
- On the other side, if a page processing cannot be commit ed because unresolvable conflicts have occurred, the scheduler should not defer a conflict notification until a user requests the commit operation.
- Read-only transactions must not affect transaction aborts for writing transactions.

Locking schedulers require the explicit locking of database objects for both reading and writing. Because we do not know when a user has finished the

work on a form field we would have to hold the locks from the begin of the transaction when the initial values are read until the commit. This could lead to a serial transaction execution and reduce parallelism drastically.

The timestamp ordering protocol evades locks by use of totally ordered, unique timestamps that are assigned to each transaction. This protocol has the drawback that among several conflicting transactions only the one with the highest timestamp successfully commits. This is not compatible with requirements two and four.

In our implementation, we have opted for an optimistic protocol (Härder, 1984; Kung and Robinson, 1981) where newly arriving operations simply pass and the burden of concurrency control is deferred until the end of the transaction. Thereby, detected conflicts are resolved by aborting transactions.

The execution of a transaction is divided into two phases, the read phase and the valwrite phase. During the read phase all read operations are logged in the readset RS and all write operations are performed in a private workspace, the writeset WS. So the written values are not visible to other transaction before a transaction successfully commits.

The valwrite phase is initiated with the commit operation. It tests whether the transaction execution has been correct in the sense of conflict serializability. If so, the writeset of the transaction is transferred to the database, otherwise the transaction is aborted.

This protocol meets best the previously formulated requirements. It is only perceivable in the case of conflicts and favours the transaction which initiates the commit first.

3.1 The FOCC Scheduler

Our scheduler protocol follows the forward oriented optimistic concurrency control presented in (Weikum and Vossen, 2001), where a transaction is validated against all concurrent transactions.

Here, a transaction t_j is validated at the time n if the following holds for all concurrent transactions t_i:

$$WS(t_j) \cap RS^n(t_i) = \emptyset$$

Thereby, $RS(t)$ and $WS(t)$ are the readset and writeset of transaction t.

Hence, the scheduler makes sure that no values other concurrent transactions have read are overwritten, and thereby outdated, by the validating transaction. This is immediately satisfied if the validating transaction is read-only.

Since these transactions are not yet committed we gain some flexibility in handling a detected conflict. In order to satisfy the second requirement we follow the 'kill and commit' scheme presented in (Härder, 1984), where a non validating transaction nevertheless commits and the conflicting transactions abort.

To sketch the proof that our scheduler produces conflict serializable schedules only we show that the produced conflict graph is acyclic.

With the commit of transaction t_j we insert the new node t_j into the so far acyclic conflict graph. If the graph became thereby cyclic the node t_j had to be involved in the cycle. Therefore, the node t_j had incoming and outgoing edges. For an incoming edge t_j would have read values that were written by an already committed transaction. That would have lead to an abort of t_j. So, only outgoing edges are possible and the conflict graph remains acyclic. □

Beside the readset and the writeset our scheduler has to hold the status of all running transactions in an internal table. Thereby, the status of a transaction is either *running*, *committed*, *aborted* or *in conflict*.

In the following we resume the actions taken by our scheduler for each possible request:

- *begin*: The scheduler generates a new unique transaction id and inserts it in the status table as *running*.
- *abort*: The readset and writeset of the transaction is deleted and the status of the transaction is set to *aborted*.
- *commit*: The transaction is validated and the writeset is transferred to the database. If the validation was not successful the status of the conflicting transactions is set to *in conflict*.
- *read*: First, it is checked whether the transaction status is set to *in conflict*. In this case the browser is signaled to abort the transaction, otherwise the value of the requested object is read from the database and the readset is updated.
- *write*: Similar to the *read* request either the writeset is updated or a conflict is signaled.

Example An example for the execution of a schedule under this protocol is shown in figure 4.

At time of the commit of transaction 1 its writeset and the readset of the only concurrent transaction 2 are disjoint, so the validation is successful and transaction 1 is written to the database.

The next transaction to commit is the second. Its writeset is only disjoint with the readset of concurrent transactions 4 and 5 while it overlaps with the readset of transaction 3 ($RS(3) = z, WS(2) = z$). So, the status of transaction 3 is set to *in conflict* and with its next operation, here $r(x)$, a conflict is signaled and consequently the transaction aborts.

The next committing transaction is the read-only transaction 5. It has an empty writeset and thus successfully validates. The validation of transaction 4 now trivially succeeds, since there are no more concurrent transactions.

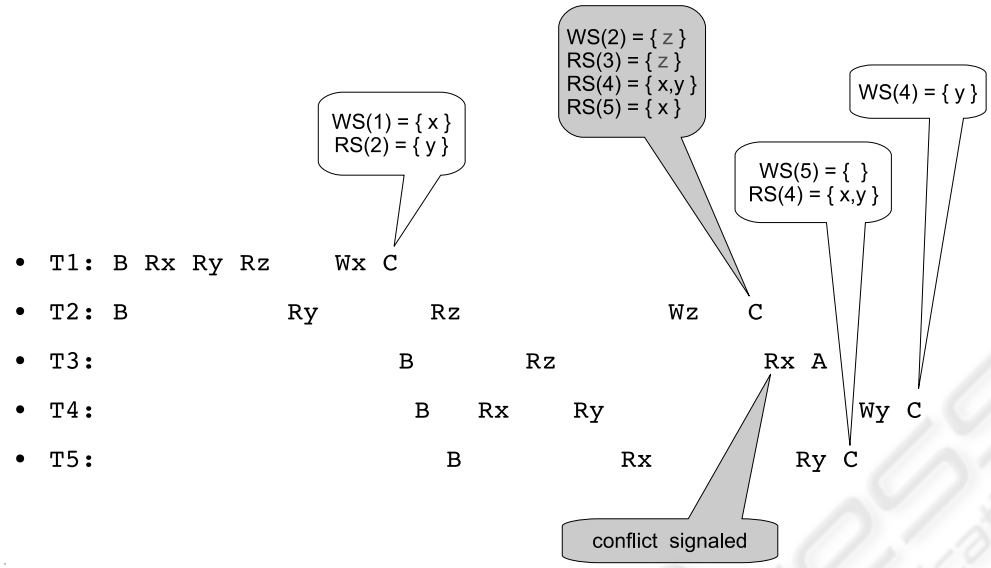

Figure 4: An Example schedule.

3.2 Distributed Transactions

As we have already seen in section 2.4, TransForm even supports distributed transactions (Bernstein and Goodman, 1981). There, several services working on different databases were addressed in one web page. Of course, global serializability must be guaranteed. Therefore, the subtransactions at each site, respectively service, either commit or abort altogether.

Fortunately, the following holds for distributed transaction over several sites (Lewis et al., 2002): If the concurrency control at each site uses an optimistic algorithm and the system uses a two-phase commit protocol, every global schedule is serializable.

The two-phase commit protocol (Lampson and Sturgis, 1979) is easily adapted in our system. We therefore just need the additional transaction control operations *prepare*, *ready* and *done*. The javascript program naturally takes the role of the coordinator.

4 RELATED WORK

Concurrency control is a well known problem in the domain of web applications.

A typical approach to solve this problem is the one taken in JPernLite (Yang and Kaiser, 1999) that supports advanced transactional features for web applications. Unlike TransForm, JPernLite is a middleware approach where an external transaction server that operates independently of web servers provides the concurrency control capabilities to client applications.

Cheetah (Pardon and Alonso, 2000) is a java-based set of tools for building composite components that interact transactionally. Its main contribution is the composite system architecture and the support for nested transactions that are not addressed in our approach.

Besides the use of external components there have been attempts to build transactional capabilities into the existing WWW infrastructure. In contrast to TransForm, in (Lyon et al., 1998) web servers must be extended to be compliant to the protocol that provides the concurrency control functionality. Then, a client can explicitly submit operations as part of transactions.

WebDav (E. James Whitehead and Goland, 1999) is an HTTP extension for web servers that intend to support locking of their web pages. In (Shadgar and Holyer, 2004) the authors make use of WebDAV to provide a methodology for accessing and authoring databases.

Though this last approach comes close to our system in addressing a database object via an URL, TransForm is to the best of our knowledge unique in that it provides transaction safe data access for web application without the need for additional components besides the web server.

5 CONCLUSION

In this paper we have presented TransForm, a novel model for web applications that specifies the access to data sources directly in the markup of a web page and executes it with the help of a javascript program in

the browser and a web service that forms the interface to the data sources.

We have shown that this data access happens in a transaction safe way. Thereby transactions span the whole lifecycle of a web page and are independent of the preceding server action if this server action generates the page without dependency on the database.

Hence, TransForm integrates well into other web application frameworks providing them with a concurrency control component.

We have presented the scheduler of our implementation, following an optimistic approach, as one of several possible scheduler strategies. Optimistic concurrency control schemes were designed under the explicit assumption that conflicts among transactions are rare events. This does not hold generally in the context of web applications. So, our scheduler may not be the best in all circumstances and therefore we plan to examine other strategies and test them under different server workloads.

Another field we are working on is the resource optimization. Currently only the transaction id is stored in the browser while the service manages the bulk of context information for all running transactions. In order to increase the server performance we try to shift parts of this context information to the browser, which in turn submits it back to the service when requesting a transaction commit.

In this paper we have mainly presented the parts of TransForm that deal with form processing. Albeit an important part, it is only one aspect of the complete framework. TransForm provides support for arbitrary services that can be included into a web page in a transaction safe way. They range from the integration of dynamically created parts of the application to the embedding of commercial off-the-shelf components (COTS).

REFERENCES

Bernstein, P. A. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221.

Ceri, S., Fraternali, P., and Bongio, A. (2000). Web modeling language (webml): a modeling language for designing web sites. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 137–157, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Draheim, D. and Weber, G. (2004). *Form-Oriented Analysis: A New Methodology to Model Form-Based Applications*. Springer Verlag.

E. James Whitehead, J. and Goland, Y. (1999). Webdav: a network protocol for remote collaborative authoring on the web. In *Proceedings of the Sixth European conference on Computer supported cooperative work*, pages 291–310, Norwell, MA, USA. Kluwer Academic Publishers.

Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1998). Hypertext Transfer Protocol – HTTP/1.1. Technical Report Internet RFC 2616, IETF. `http://www.ietf.org/rfc/rfc2616.txt`.

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures.

Garrett, J. J. (2005). Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

Härder, T. (1984). Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120.

Härder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317.

Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226.

Lampson, B. W. and Sturgis, H. E. (1979). Crash recovery in a distributed data storage system. Technical report.

Lewis, P. M., Bernstein, A., and Kifer, M. (2002). *Databases and Transaction Processing*. Addison-Wesley.

Lyon, J., Evans, K., and Klein, J. (1998). Transaction Internet Protocol. Technical Report Internet RFC 2371. `http://www.ietf.org/rfc/rfc2372.txt?number=2372`.

Pardon, G. and Alonso, G. (2000). Cheetah: a lightweight transaction server for plug-and-play internet data management. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 210–219, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Shadgar, B. and Holyer, I. (2004). Adapting databases and webdav protocol. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 612–620, New York, NY, USA. ACM Press.

W3C (1999). XML Path Language (XPath) Version 1.0. Technical report.

W3C (2000). Simple Object Access Protocol (SOAP) 1.1. Technical report.

Weikum, G. and Vossen, G. (2001). *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Yang, J. and Kaiser, G. E. (1999). Jpernlite: Extensible transaction services for the www. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):639–657.