Web e-Transactions*

Svend Frølund Fernando Pedone Jim Pruyne

Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
{frolund, pedone, pruyne}@hpl.hp.com

Abstract

An e-transaction is a powerful abstraction that ensures exactly-once semantics in three-tier applications. It is very common for applications on the Internet to follow a three-tier architecture, and we discuss protocol and implementation issues for e-transactions in the Internet setting (so-called web e-transactions). In contrast to current related protocols, which require some sort of retry logic in the client-side software, our protocol assumes a standard web browser and only uses the semantics of HTML and HTTP to implement the required client-side retry logic. We discuss how to implement the server side of our protocol in the context of JAVA servlets.

1 Introduction

With electronic commerce over the Internet, the business is closed if the web-site is down. Thus, it is very important for Internet applications to be highly available. The common architecture for Internet applications is a three-tier system: thin frontend clients (browsers), stateless middletier servers (web-servers), and backend databases. High availability in such systems is typically addressed through two different kinds of clustering. Web servers run in a web-server farm. Because of the stateless nature of HTTP, and of the web servers themselves, any web server in the farm can serve any request. Databases run in high-availability clusters, which is clustering at the hardware level. A set of nodes in a cluster have access to a shared disk, and the clustering software monitors the database processes and restarts them as necessary (possibly after failing over to another node in the cluster).

Although the use of clustering reduces down time, it does not address the issue of failure transparency. For example, consider the situation where a web server is executing a transaction against the backend database in response to an HTTP request from a browser. If this web server crashes, the browser (and thereby the end user) will receive an error message, even if other web servers are still running. If the transaction was supposed to update state in the database (purchase tickets, transfer money, and so on), the user is now left wondering what actually happened—whether the update took place before the crash.

Current web-based transaction-processing systems typically cover the following spectrum in terms of failure handling:

- Users are exposed to failures and given no assistance in handling them.
- Users are exposed to failures and given warnings about what not to do.

In the latter case, for example, when the user submits a form to start a transaction, the web server may return a page with a warning, informing the user of the dangers of re-submitting the form. The user is not empowered with safe retry mechanisms, but instead told to contact customer

*© CSREA. This is a revised version of a paper, with the same title, that appears in the proceedings of the Internet Computing Conference, Las Vegas, June 25–28, 2001.

service in case of failures. The user is essentially "empowered" with a transaction id, to give to customer service, instead of retry logic.

Our aim in this paper is to isolate users from the handling of errors. Abstractly speaking, isolation from errors is equivalent to exactly-once semantics for the user-initiated transaction. Thus, we can re-phrase our aim as providing e-transactions (exactly-once transactions) for the web environment. We provide a large degree of automatic retry where the user is not involved in the error handling at all. Furthermore, in the situations where retry is not automated, we empower the user with safe retry. With safe retry, errors are visible to users, but users can determine the outcome (commit or abort) of the transaction through interaction with the web site. For example, the user may follow a link to an "outcome determination" page where the outcome can be retrieved.

There is a direct correlation between how much one can isolate users from error handling and the expressiveness of the error-handling logic in the browser. For example, if we assume that browsers download and run applets, which then access middle-tier application servers via protocols such as IIOP or RMI, we can inject arbitrary retry logic in the stub that the applet uses to communicate with middle-tier servers. This is the approach taken in [FG01, FG00]. With arbitrary client-side logic, we can completely isolate users from errors. In this paper, we consider the very common case where browsers download web pages only. That is, we assume that the protocol between browsers and servers is HTTP, and that the content being downloaded is standard HTML. Moreover, we do not want to rely on browser plug-ins since many users are reluctant to install them. By considering pure HTTP and HTML only, we can apply our mechanisms very broadly, but we cannot completely isolate users from error handling. The intuitive reason that we cannot obtain complete transparency is that the error-handling logic is essentially embedded in downloaded pages. This means that there is a "vulnerability" window from the time a form is submitted until the page with error-handling logic is received by the browser.

Considering pure HTTP and HTML imposes certain constraints:

- The client-server protocol is a pure request-reply protocol. That is, the server cannot notify the client; the server only responds to client requests.

- Execution of logic by the client is visible to the user. For example, if we want the client to send a request to the server to check if a transaction is still in progress, the browser would have to request a web page from the server, and this web page would be rendered by the browser. Thus, we have to be judicious about how often we issue requests.

We present a protocol that ensures e-transaction semantics, except in a small number of failure cases. In those cases, the protocol ensures at-most-once instead of exactly-once. The protocol uses pure web technologies, such as HTTP and HTML, and takes the above constraints into account. Thus, the protocol can be used with standard browsers, and with the standard content-delivery methods.

We first present the idea behind our protocol as abstract pseudo-code in Section 2. We then show how to implement the protocol in the context of the very common JAVA servlet programming model. We present this implementation in Section 3. Finally, in Section 5, we discuss our approach and draw our conclusions.

2 Protocol

2.1 Assumptions

We assume a system with standard web servers and web browsers. The browsers request pages, in HTML, from the web servers using the HTTP protocol. Web servers can fail by crashing, that is, they execute their prescribed behavior until they crash—we do not consider Byzantine failures. Any available web server can treat a request from a web browser. To execute browser requests, web servers run transactions against a shared database. We do not explicitly consider availability of the database in this work. We simply assume that a database will eventually recover from any failure either using standard availability clustering techniques or with a more sophisticated mechanism such as the one described in [PF00].

In order to ensure exactly-once semantics, we rely on the browser to stay up until the transaction finishes its execution—this is because the transaction retry logic is driven by the browser. This does not imply that a browser crash can lead to arbitrary behavior by the system. If a browser crashes, our protocol ensures at-most-once semantics. In Section 5, we discuss how to extend our scheme to cope with browser recovery, that is, how to enforce exactly-once semantics even in the event of browser crashes.

We assume that at any time there is at least one web server up and available to process requests. Moreover, we assume that the web servers run as part of a "web-server farm" that is accessed through a single DNS entry, and we assume that the DNS name resolution will eventually resolve the name of the web-server farm to an operational web server. This means that if the web server being used by a web browser crashes, the browser will eventually resubmit the request, which will be delivered to an operational web server. This DNS-based approach is a standard method for handling web-server farms.

In response to certain page requests, a web server may start a transaction against a backend database. We assume that the database allows XA-style [x/O91] transaction control. We can use a `start` method to initiate transactions, and `commit` or `abort` methods to terminate the transaction. We augment the XA-based termination with the notion of a *testable* transaction as defined in [FG00]. Essentially, a testable transaction is one whose outcome (commit or abort) and result (the value which is produced by the transaction's SQL statements) can be determined in a fault-tolerant and highly-available manner. For the very common case of a single backend database, the testable interface can be layered on top of the XA transaction handling mechanism. In

```
interface testable {
  Outcome commit(Result,UUID);
  void rollback(UUID);
  Result get-outcome(UUID);
}
```

Figure 1: The signature of testable transactions

Section 5, we discuss how to implement the notion of testable transaction on top of a single, standard database.

The functionality of testable transactions is available through the interface in Figure 1. The `commit` method tries to commit a given transaction with a given result. The `get-outcome` method takes a transaction identifier and returns the result for that transaction, if any. If `get-outcome` returns nil, no result has been committed yet for that transaction. The `rollback` method simply terminates a given transaction without committing it. Transaction identifiers are global: one web server can call `commit` and another web-server can call `get-outcome` for the same transaction. Moreover, we require the outcome and result information to be highly available in the sense that one web server can commit a transaction, and another web server can determine the outcome and result of the transaction independently of the first web server, that is, even if the first web server crashes.

To provide exactly-once semantics for transactions, we assume that any transaction will eventually be able to run to completion and generate a result. The database is allowed to fail an arbitrary number of times, but must eventually stay up long enough to execute transactions to completion.

2.2 Failure-Free Execution

In Figure 2, we illustrate the protocol's basic interaction by considering a run without failures. Since the protocol is based on HTTP, it consists of a number of request-reply interactions. In the figure, these interactions are demarcated by dashed

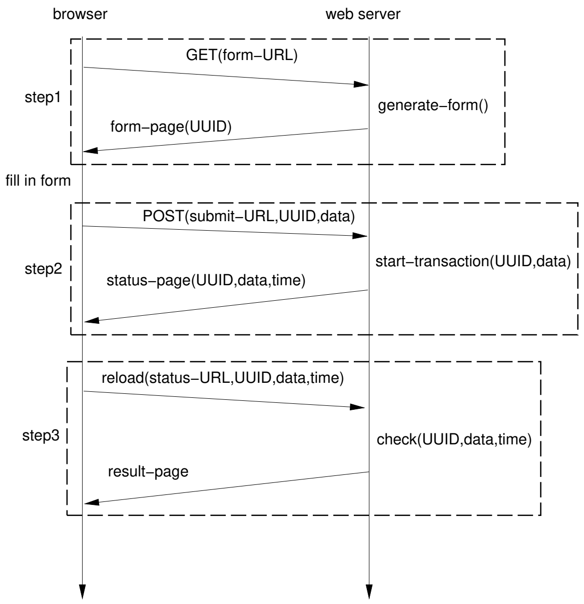

Figure 2: A run of the protocol without failures

boxes, and denoted as `step1`, `step2`, and `step3` respectively.

The first step of the protocol is for the client to request the web page, which contains the form to be submitted with exactly-once semantics. The URL of this form is `form-URL`. We show the server-side logic in Figure 3 as pseudo code. To serve up the page with `form-URL`, the server first generates a globally unique identifier (UUID), and then embeds the generated UUID in the HTML returned to the browser. (Notice that the HTML "form-page(UUID)" returned to the browser in Figure 3 is generated by the "form-html(uuid)" method.) The UUID will be used as an identifier for the server-side transaction which processes the form data. The idea is for the browser to pass this UUID back to the server when the browser submits the form. To make this work, we can either store the UUID in an in-memory cookie in the browser, or we can store the UUID in the URL that the client uses to submit the form (this URL is part of the server-generated form). In general this UUID can be handled similarly to the session identifiers that web servers often use today.

When the browser receives the form page with the embedded UUID, it renders the page, and the user can now fill in the form data as usual. When the user has entered the form data, he pushes a submit button in the form to send the form data to the web server for processing. The server-side processing is transactional, and this is the transaction that we want to give exactly-once semantics. When the user pushes the submit button, the browser sends the filled-in form to the web server with an HTTP POST request. In response to this request, the server executes the `start-transaction` method. This method asynchronously launches a transaction to execute the business logic (the business logic is described in the method called `biz-logic`). After launching the transaction, the server returns a so-called status page to the browser.

The status page informs the user that the transaction is in progress. The status page is set to automatically reload after a few seconds using the appropriate HTML tag. Reloading the status page, whether automatically or by the user issuing a page refresh, executes the server-side method called `check`. This method checks the status of the server-side transaction.

To properly identify the transaction, the status page contains the UUID for the transaction. When the status page is reloaded, this UUID is passed back to the server. In addition, the status page contains the form data so that the transaction can be retried if the previous incarnation is determined to have failed. Finally, the status page contains a timestamp that the server can use to determine if a certain timeout period has elapsed since the transaction was launched. By causing the (automatic) reload, the browser checks the status of the server-side transaction without the user being involved.

The logic of the `check` method is the following. If the transaction has been active for less than a given timeout value, the `check` method simply invokes the `get-outcome` method to see if the transaction has produced a result. If `get-outcome` returns nil, the transaction may still be active or it may have crashed before committing. Without calling `rollback`, we cannot distinguish between the two situations. If we call `rollback`, we can eliminate the first case (transaction still active) because `rollback` will cancel the transaction. However, we do not want to call `rollback` too aggressively: we want to give the transaction time to execute before cancelling it. Thus, we use the timestamp to determine if we simply check (by calling `get-outcome`) or check after cancelling (by first calling `rollback`). If we know that the transaction did not commit, and will not commit because we have cancelled it, we retry the transaction.

In Figure 2, we assume that the transaction has committed and stored its result in the testable transaction abstraction. Thus, the first call to `get-outcome` returns a non-nil result, which is then returned to the browser as part of a result page.

2.3 Failure Handling

Getting the form from the server (`step1` in Figure 2) is idempotent, and does not require any failure handling beyond reload by the user.

If the browser uses a web server that fails during the second step, we have to distinguish between the following two cases: (1) the browser receives the status page and (2) the browser does not receive the status page. In (1), the reload logic for the status page will perform the check against another web server. Here, the fail-over from one web server to another is taken care of by the DNS resolution against the cluster of web servers. In (2), the browser will eventually time out and display an error message to the user. In this situation, we need to involve the user in the failure recovery somehow. One way is to embed the URL of the status page in the form as a link, and instruct the user to follow that link in case of errors. These instructions also have to be part of the form.

If the third step fails, the user can simply reload the status page manually. Reloading the status page n times has at-most-once semantics because the server-side logic in the `check` method will only launch a new transaction if all previous attempts have failed.

```
html generate-form() {
  uuid := new UUID();
  return form-html(uuid);
}

void biz-logic(uuid,data) {
  start(uuid);
  // Execute SQL with data as argument.
  // Store result in the variable res
  testable::commit(res,uuid);
}

html start-transaction(uuid,data) {
  // Spawn new thread to execute biz-logic
  // with uuid and data as arguments
  time := current-time();
  return stat-page-html(uuid,data,time);
}

html check(uuid,data,time) {
  if current-time() - time > timeout then
    testable::rollback(uuid);
    res := testable::get-outcome(uuid);
    if res == nil then
      return start-transaction(uuid,data);
    else
      return result-page-html(res);
  else
    res := testable::get-outcome(uuid);
    if res != nil then
      return result-page-html(res);
    else
      return stat-page-html(uuid,data,time);
}
```

Figure 3: The server-side logic

3 Applying Web e-Transactions to Java Servlets

The Web e-Transaction protocol was designed with the web based three-tier environment in mind. We have built a prototype of the protocol using Java Servlets [Sun01a]. Servlets are a server-side programming model for executing logic in response to web-based HTTP requests, similar to the Common Gateway Interface (CGI). Servlets are provided with the parameters to the HTTP request via a `HttpServletRequest` and produce output including an HTML stream to be rendered by the browser via a `HttpServletResponse` object. We chose this platform because it closely matches our assumed, three-tier environment, is widely available, and is extremely popular. The prototype is intended to validate our assumptions about the three-tier environment.

In implementing the Web e-Transaction system, we have had two design principles:

- Minimize changes to existing code. We hope to introduce the new protocol without making significant changes to code already developed for the servlet programming model.
- Minimize the vulnerability window. As discussed in the protocol description, there are intervals in which a failure on the server will require the user to re-initiate an operation. We strive to keep these intervals as small as possible to reduce the likelihood that a user must become involved in recovery.

Figure 4 is a high-level outline of the servlet based prototype. While obviously not complete, this example is intended to show how the protocol can be adapted to the servlet model without significant changes to the abstract implementation in figure 3. We also use this example to demonstrate our design principles. For purposes of discussion, we do not use the full Java syntax, and remove many details such as synchronization points which are important, but confuse the discussion.

```
void FormServlet(HttpServletRequest req,
                 HttpServletResponse resp) {
  request.getSession(true);
  // Normal Form Creation processing
  // with Form handling servlet name stored
  // in a hidden field,
  // and action set to the Start servlet
}
void BizLogic(HttpServletRequest req,
              HttpServletResponse resp) {
  WeTDriver.getConnection(''JDBC-URL'');
  // Execute JDBC commands and other logic
  // Output result normally
  // Do NOT commit any JDBC updates
}
void Start(HttpServletRequest req,
           HttpServletResponse resp) {
  WorkQueue.queue(req.clone(),
                  resp.clone());
  SendStatPage(req, resp);
}
void Check(HttpServletRequest req,
           HttpServletResponse resp) {
  result = getOutcome(req.getSession());
  if (result != null) {
    result.send(resp);
    return;
  }
  job = getJobFromQueue(req.getSession());
  if (currentTime() - job.time > timeout) {
    job.abort();
    WorkQueue.queue(job.req, job.resp);
  } else {
    SendStatPage(job.req, job.resp);
  }
}
void WorkerThread() {
  while(true) {
    job = getJobFromQueue();
    servlet = job.targetServlet;
    try {
      // Begin Transaction
      servlet.service(job.req, job.resp);
      storeOutcome(job.req, job.resp);
      removeJobFromQueue(job);
      // Commit Transaction
    } catch (Exception e) {
      // Abort transaction
    }
  }
}
```

Figure 4: Servlets implementing the protocol

The first two methods, `FormServlet` and `BizLogic` are intended to represent the servlets for generating and processing the form respectively. These servlets are created as part of the web application, and are independent of the web e-transaction protocol. In general, we assume they have been developed prior to the introduction of the protocol, so we are concerned with minimizing changes to them.

We place two requirements on the `FormServlet`. First, it must create a session for the user. This session is used to identify the user's operations in the same way the uuid is used in the protocol description. This is a common practice, most Java servlet based systems will already do this. Second, the generated form must be changed so that the protocol handling logic is invoked rather than directly invoking the form handling `BizLogic` servlet. Our approach to this is to change the form servlet to invoke our `Start` servlet. Further, we embed the name of the `BizLogic` servlet as a hidden field in the form so that our `Start` servlet knows the proper logic to execute to handle the form.

The `BizLogic` servlet must be changed only in how it interacts with the database tier. In this example, we assume that JDBC [Sun01b] is being used to send SQL statements to the database. The protocol requires that we commit the result of the servlet call (which we assume to be the `HttpServletResponse` object at the end of the method) atomically with the updates performed by the business logic. There are a variety of approaches to solving this problem. The approach used here is to obtain a database connection from a pool managed by the protocol implementation via a call to `WetDriver.getConnection`. Upon completion, we are able to determine which connection was used by the call (e.g., by storing some information in thread local storage), and commit or abort the transaction as needed. The servlet will not perform the commit operation. Instead, it will be done on behalf of the servlet as we show below.

The remainder of the example is new logic introduced which drives the protocol. This needs to be implemented only once, and can be applied to an arbitrary number of form generating and business logic servlets. The `Start` servlet is called when a form is received from the client browser. Its job is to quickly queue the incoming form data and return the status page. This must be done quickly because the interval during which the servlet is running constitutes one of our vulnerability windows. We also must insure that the form data cannot be lost. There are two approaches to solving this problem. One is to embed the form data in the status page returned. This creates a larger HTML stream to be returned, and incurs some processing overhead. The other is to store the form data in a persistent location, such as the database, as part of the queuing operation. Either approach is possible, but avoiding interactions with the database seems to minimize the vulnerability window. The status page generated forces the browser to automatically invoke the `Check` servlet after a short delay via the HTML meta-operation "refresh." The algorithm of the `Check` servlet is identical to the check method of figure 3. We use the session to search for outcomes in place of the UUID used in the previous code.

We assume that one or more instances of the `WorkerThread` are always running in the background.[1] These threads simply remove service requests (jobs) from the work queue, and invoke the servlet associated with the request. The most important role of the worker threads is to insure that transactions are properly committed or aborted. As discussed previously, this can be done by tracking the use of JDBC connections. Before invoking the servlet, a transactional context is initiated. If the servlet executes normally (i.e., no exceptions are thrown), the thread stores the result to the database, and commits both the result and the servlet's operations. In the event of an exception, the transaction is aborted. In the event of a servlet failure we could either requeue the job to be tried again, perhaps with a maximum number of re-tries, or generate a re-

[1] In the abstract implementation, we describe creating a thread for each request as it arrives. The prototype uses a pool of threads for efficiency, but the methods are otherwise equivalent.

sult signaling a permanent failure to the user. If we do generate such a failure, the user has the added benefit of knowing that no operations of the transaction were committed.

4 Related Work

Traditional web-based applications use backend transaction monitors [BN97] to provide reliability. Transaction monitors ensure that application servers update backend databases atomically (i.e., with all-or-nothing semantics). Although transaction monitors keep the backend state consistent, they do not provide end-to-end reliability: the atomicity guarantee does not include the frontend browser. The goal of [LS98] is to extend the backend atomicity semantics to also cover client-side state, such as cookies or other goods that the client has purchased from the service. The main idea is to use a notion of resource proxy to make the client transactional without registering it with the backend transaction monitor. In contrast to our approach, [LS98] relies on the downloading of applet code to the browser. Moreover, the atomicity guanrantee requires a two-phase commit protcol, even if there is only a single backend database (in [FG00], we show how to implement the testable transaction abstraction with a one-phase commit protocol for a single database).

Rather than end-to-end atomicity (all-or-nothing), we focus on masking backend failures to frontend clients. Where atomicity gives at-most-once semantics, we are interested in exactly-once semantics. We defined the notion of e-transaction, and developed protocols to implement e-transactions in various settings [FG00, FG01]. Unlike those protcols that assume arbitrary retry logic at browsers, the protocol in this paper relies on standard HTML handling only. Another way to provide exactly-once semantics in three-tier systems is through message queues [BHM90]. For web-based applications, this would require client requests to go through message queues. Besides requiring significant rearchitecting of existing web-based applications, this also would incur a performance penalty: storing requests in message queues gives rise to additional disk writes in the critical path. Furthermore, using message queues would require a server-side two-phase commit protocol, even with a single backend database because the queues are now transactional resources.

5 Discussion

This paper discusses implementation issues about web e-transactions, a powerful abstraction to build e-transactions on the Internet. In contrast to current related protocols, which require some sort of retry logic in the client-side software, our protocol assumes a standard web browser and only uses the semantics of HTML and HTTP to implement the required client-side retry logic.

As part of our protocol, after the browser submits a form, it quickly receives a "transaction in progress" page in return from the web server. This operation needs to be as fast and simple as possible because if the server should fail during this operation, the user will be required to repost the form manually. The process of quickly returning a transaction in progress page is already used by many e-commerce sites, and should not be terribly jarring to users.

Our protocol also relies on the notion of testable transactions. With a single backend database, we can implement a testable transaction by storing the transaction result in the database itself. This approach is outlined in [FG00]. The basic idea is for the `commit` method in the testable abstraction to execute SQL statements within the pending transaction, which inserts the result and transaction identifier into a special "log" table in the database. We can then implement `get-outcome` in terms of queries against this table. Because the table insertion is performed within the pending transaction, the storage of the result and the transaction commit are atomic.

If the transaction committed, the result is sent to the browser. Returning the result to the user could fail, however, so it is not safe to immediately remove the result from the database. The result information store in the testable transaction abstraction will have to be garbage collected as described in [FG00].

Finally, we rely on browser liveness during the execution of a transaction to extend at-most-once to exactly-once. If a browser crashes, it obtains at-most-once semantics. We can extend our scheme to facilitate browser recovery in the following way. When the browser receives the form in the first step of our protocol, the server could ask the browser to store the UUID in a cookie. The UUID is then persistent, and can be used if the browser recovers after a crash. That is, the web site can provide a recovery page that users can go to after a crash. When loading the recovery page, the browser sends the cookie with the UUID to the server, and the server can then use the outcome determination of testable transactions to figure out what happened and instruct the user accordingly.

Our future work will be to continue to validate the protocol via prototyping. Our initial prototype has demonstrated the validity of the protocol in simple cases, but we need to measure the vulnerability windows, and continue to look for ways to minimize it to ensure that users are not required to be involved in failure recovery. We also would like to find more seamless ways to integrate web e-transactions into existing web application environments. This may include incorporating other technologies, such as Enterprise Java Beans, into the implementation. Because the logic for the protocol is independent of the logic of the applications, it should be possible to move it into the web servers directly. This should reduce the visible changes to existing web applications. For example, they would no longer be required to change the name of servlets they invoke.

References

[BHM90] P. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, May 1990.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan-Kaufmann, 1997.

[FG00] S. Frølund and R. Guerraoui. A pragmatic implementation of e-transactions. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, October 2000.

[FG01] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2), February 2001.

[LS98] M. C. Little and S. K. Shrivastava. Integrating the object transaction service with the web. In *Proceedings of the Second International Workshop on Enterprise Distributed Object Computing (EDOC)*. IEEE, 1998.

[PF00] F. Pedone and S. Frølund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2000.

[Sun01a] Java servlet technology. http://java.sun.com/products/servlet/index.html, 2001.

[Sun01b] Jdbc technology. http://java.sun.com/products/jdbc/index.html, 2001.

[x/O91] x/Open Company Ltd. *Distributed Transaction Processing: The XA Specification*, 1991. XO/SNAP/91/050.