E-Commerce with Rich Clients and Flexible Transactions

Dylan Clarke, Graham Morgan
School of Computing Science, Newcastle University
{Dylan.Clarke,Graham.Morgan}@ncl.ac.uk

Abstract

In this paper we describe an approach for implementing a shopping cart program using rich clients. We assume our client is not always connected to the server side during a purchase. We utilize the well known approach known as flexible transactions to afford a best effort approach to successfully complete a purchase order. Our approach is timely as commercial solutions for rich client technology (such as adobe air) is now a realistic proposition for many Internet application developers.

1. Introduction

The assumption of full connectivity and limited client functionality has been viewed as a hindrance in recent years for end user Internet applications. This is because any enrichment of the user experience relates to increasing server loads and high bandwidth requirements. In addition, the latency present between client and server increases the wait time for users.

Rich clients are being proposed as a way to provide as much functionality as possible to a user while limiting dependency on the server side. A user may work while disconnected from the server, with interaction between client and server handled appropriately whenever connectivity is available.

Rich clients go beyond browsers with enhanced functionality; they represent development platforms themselves, with persistent store capabilities and an expectation they will run continuously in the background. In this scenario a web browser becomes just another program that runs within a rich client.

Considering the increasing popularity of rich clients one may reconsider quite fundamental implementations of existing web based applications. One such application is the shopping cart. In this paper we describe our initial experiences of implementing a rich client shopping cart using flexible transactions on the adobe air platform. Instead of the immediate requirement of all-or-nothing, flexible transactions allow for a more forgiving environment, where sub-transactions may be compensated for at a later time if the need arises.

Section 2 describes background and related work. Section 3 describes our approach and section 4 presents conclusions drawn from our work.

2. Background and Related Work

In this section we first identify the scope of our work and then present an introduction to different transactional approaches. We then briefly describe rich clients. We indicate why our transactional approach is suitable in the context of rich clients and what benefits this may bring over the non-rich client approach.

2.1 Scope

Although our overall project covers all the areas of catalogue updates, user login and security, we concentrate on the last steps of the purchase process here to afford a detailed description of our transactional approach. As client connectivity to the server may be intermittent in nature, we do not wish a transaction to fail because a client is not connected to the server for a period of time.

Worth mentioning at this point is that we could utilize Web Service transactional services (e.g., [9]). However, the substantial cost in middleware overhead in adobe air (and the lack of implementation) meant we implemented our own techniques. As the main focus of the paper relates to a shopping cart and not a general solution, we consider this approach adequate.

2.2 Flexible Transactions

The ACID properties of a standard transaction cause difficulties for long running activities as timeouts will inevitably occur [7]. If we allowed timeouts to be substantial, measured in hours or days, the problem

becomes one of hindering the overall performance of an application. This is due to the fact that the locking of resources (possibly over a number of different transactional participants) may block forward progression. Alternatively, high abort rates may occur as interference between resources is detected that conflict with the atomicity requirement. An approach to solving this problem and still maintaining some transactional benefits is via the use of advanced transaction models. One such model is the flexible transaction model.

A flexible transaction model allows participants to work independently of each other, allowing a series of separately operating transactions to be identified as a single transaction [8]. Such transactions are viewed as sub-transactions and cumulatively represent a flexible transaction; a flexible transaction is a partial ordering of sub-transactions. Each sub-transaction may be classed as compensatable, retriable or pivot [9]. Compensatable sub-transactions can be "undone" in the sense that once committed can be compensated for by enacting a compensatable transaction. A retriable sub-transaction may be retried one or more times until it eventually commits. A pivot sub-transaction is neither compensatable nor retriable.

A major consideration in e-commerce solutions is to ensure exactly once semantics [6]. Therefore, care must be taken when developing any multi-participant transactional approach that all required state changes occur as expected, without duplication or partial loss of state.

3. Transactional Approach

In this section we first state our assumptions relating to the implementation environment and behavior of the different parts of our implementation. We then provide an overview of how our flexible transaction works without and with compensation.

3.1 Assumptions

We assume that three participants exist in our scenario: (i) ***Client*** – a rich client accessed by a user; (ii) ***Payment server*** – a server responsible for managing credit balances of users; (iii) ***Stock server*** – a server responsible for managing stock dispatch.

Each participant must have access to a local database capable of carrying out transactions (all transactions are considered retriable). In addition, each participant will be capable of completing their roles given sufficient time to do so and local databases are always accessible (failure may occur but as some point participants will function correctly and committed data is never lost). Although connectivity may be transient between the participants, we assume that connectivity will occur sufficiently long enough at some point in time to allow message passing. Furthermore, we assume that participants may not, unilaterally, decide that a transaction they previously committed may be compensated. For example, the payment server may not commit a payment transaction, say T, and then, with no interaction with client or stock server, decide to compensate T. Malicious behavior of participants is not considered. Finally, for the purposes of clarity a client is associated to only a single purchase order at a time.

3.2 Without Compensation

Figure 1 shows a progression without compensation and should be used to aid in understanding our description. When a user indicates that they wish to purchase the items stored in their shopping cart a transaction is started in the client (T_c^{order}). In this transaction a client writes order details to the database at the client side. These details include a client generated purchase number (P_n) which is used to uniquely identify this sale across all servers and the items purchased themselves. Within T_c^{order} the client sets a flag ($F_c^{payment}$) as FALSE, indicating that this particular purchase is yet to be paid for. In addition, another flag (F_c^{stock}) is set to FALSE to indicate that stock is yet to be dispatched. If T_c^{order} successfully commits then details relating to the value of the purchases are sent to the payment server in message $M_c^{payment}$ and purchase details are sent to the stock server in message $M_c^{purchase}$.

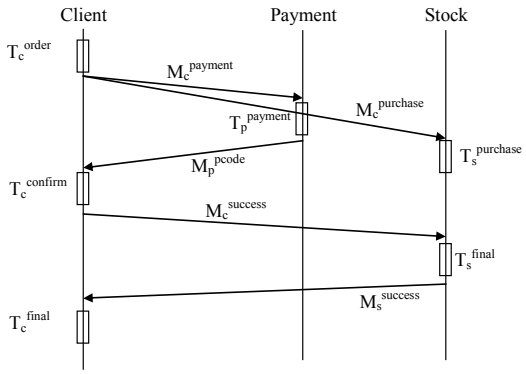

Figure 1 – Non-compensated purchase

When the payment server receives $M_c^{payment}$ a transaction to handle the required payment procedure is started ($T_p^{payment}$). The payment server first checks to determine if P_n already exists in its own local database.

If P_n is not already registered in its database then the payment server debits the customer account by the appropriate figure and readies a payment confirmation code to be stored in its database and to be returned to the client (M_p^{code}). Once $T_p^{payment}$ has completed M_p^{code} is sent to the client. If P_n already exists when $M_c^{payment}$ is received then the appropriate M_p^{code} is resent.

When the stock server receives order details from the client it first checks its database to determine if P_n already exists. If P_n does not exist then the stock server starts a transaction $T_s^{purchase}$ and stores details relating to the purchase in its local database and sets the flag $F_s^{purchase}$ to FALSE, indicating that the purchase is yet to be paid for.

When the client receives M_p^{code} from the payment server a transaction is started $T_c^{confirm}$ to set $F_c^{payment}$ to TRUE and the client sends a message to the stock server indicating that payment has been successful ($M_c^{success}$).

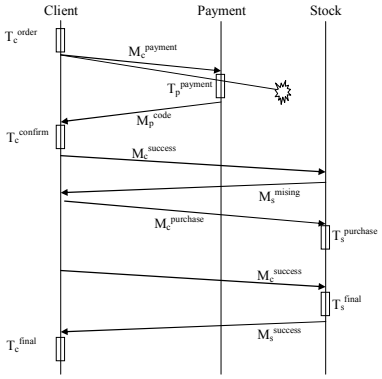

Figure 2 – Missing purchase at stock server

On receiving $M_c^{success}$ the stock server first checks to determine if P_n has already been recorded in its local database. If it has not then the stock server sends a message to the client $M_s^{missing}$ indicating that no purchase record exists regarding this P_n. This is shown in figure 2. Note that this process may be repeated a number of time until $T_s^{purchase}$ can be achieved. If P_n does exist then a transaction T_s^{final} is used to set the appropriate $F_s^{purchase}$ flag to TRUE and so indicate that stock dispatch is required. A message $M_s^{success}$ is then sent to the client, to confirm that stock dispatch has occurred. If $F_s^{purchase}$ is found to be TRUE during T_s^{final} then the transaction aborts as the stock has already been dispatched to the client and $M_s^{success}$ is resent to the client.

On receiving $M_s^{missing}$ from the stock server the client resends $M_c^{purchase}$. On receiving $M_s^{success}$ the client sets the F_c^{stock} flag to TRUE by starting and completing the T_c^{final} transaction. Once $F_c^{payment}$ and F_c^{stock} are both TRUE then the flexible transaction is considered complete.

A client makes a decision on when to send messages based on the value of the $F_c^{payment}$ and F_C^{stock} flags and if these are resends then such messages can be achieved without user intervention. If one or both flags are in the FALSE state then the following is attempted by the client: (i) $F_c^{payment}$ is set to FALSE then $M_c^{payment}$ is issued to the payment server and $M_c^{purchase}$ is sent to the stock server; (ii) $F_c^{payment}$ is TRUE yet F_c^{stock} is FALSE then $M_c^{success}$ is sent to the stock server; (iii) if $M_s^{missing}$ is received then $M_c^{purchase}$ is sent to stock server. We rely on the payment server and the stock server replying to client messages and do not require these servers to initiate message passing, only to respond to client messages.

3.3 Compensation

There is a possibility to compensate some transactions at all participants if both $F_c^{payment}$ and F_c^{stock} are not TRUE. However, we have deemed two transactions as non-compensatable: (i) T_c^{final}, where we recognize the end of the purchase at the client; (ii) T_s^{final}, where we realize stock dispatch at the stock server. We assume that recalling stock is logistically unappealing. This leaves us with the following compenstable transactions: T_c^{order} (recording client order at client); (ii) $T_p^{payment}$ (recording of payment at payment server); $T_s^{purchase}$ (recording of purchase details at stock server); $T_c^{confirm}$ (recording of payment acknowledgement at client).

If T_c^{order} fails to commit then the whole process ceases and no messages are ever issued to the payment and stock servers. In such a scenario a user may be informed that their purchase failed immediately. However, if T_c^{order} commits the client then assumes the role of attempting to complete the purchase process via the repeated sending of messages to the payment and stock servers as and when appropriate given transient connectivity (as mentioned in previous section). As one of our assumptions is that at some point messages will exchange and transactions commit, we rely on unfavorable message responses from the payment and/or stock servers to compensate already committed transactions. An unfavorable message may be considered application dependent and relates to the scenarios “out of stock” or “insufficient finances”. Therefore, our discussion centers on two possible scenarios: (i) the payment server is unable to carry out the financial transaction; (ii) the stock server is unable to satisfy the sale request.

To accommodate compensation we introduce two additional flags at the client. These two flags are used to indicate when there is an inability to carry out the purchase at the payment server (F_c^{pfail}) or the stock server (F_c^{sfail}). Each server has its own failure log that keeps track of failed and compensated transactions (payment – L_p^{fail}, stock - L_s^{fail}). The failed logs are used to prevent the payment and stock servers from acting on messages from a client repeatedly (possibly carrying out compensation erroneously more than once).

We now present a sketch of our approach. The diagrams are presented to ease understanding and do not present all possible scenarios.

3.3.1 Payment Server. Figure 3 provides an overview of the compensation process when the payment server is unable to honor its obligations. On receiving $M_c^{payment}$ the payment server may deem the transaction $T_p^{payment}$ unwarranted due to problems with a client's account (e.g., does not exist, unrecognized client details, suspended account). In such circumstances the payment server first determines if this purchase (P_n) has been acted upon previously. If P_n exists in the payment servers own database then the payment server assumes that, although invalid at this attempt, a previous attempt at the purchase succeeded. Therefore, actions of the payment server follow the same progress as described without compensation (no compensation). If P_n does not exist in the payment server's database nor the payment server's failed log then the payment server records P_n using the transaction T_p^{fail} in L_p^{fail} and issues a message M_p^{fail} to the client if the T_p^{fail} succeeds. If P_n does exist in the failed log already then M_p^{fail} is resent to the client.

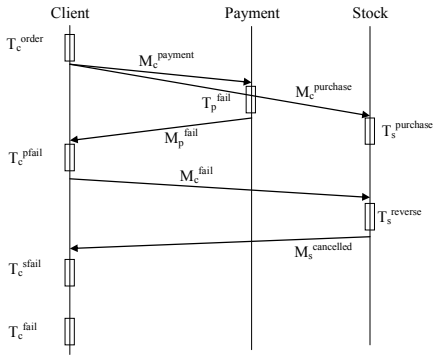

Figure 3 – Compensating in the presence of payment problems

When the client receives M_p^{fail} from the payment server there is a requirement for the client to inform the stock server that the previous request for stock purchase is now not needed. First, the flag F_c^{pfail} at the client side is set to TRUE using transaction T_c^{pfail} to indicate that failure of the payment server to honor the sale has been noted. Once F_c^{pfail} is TRUE the client sends the message M_c^{fail} to the stock server. Like other client messages, this message will be sent periodically as long as F_c^{pfail} is set to TRUE and the F_c^{sfail} flag is set to FALSE.

When the stock server receives a M_c^{fail} message from the client it checks to see if P_n exists in the local database. If P_n does exist then the stock server reverses the stock allocation of the purchase and deletes the original purchase data entry created by the original purchase transaction and records that P_n is associated to a failed transaction in L_s^{fail}. This is achieved in the single transaction $T_s^{reverse}$. Once $T_s^{reverse}$ has completed the stock server sends $M_s^{cancelled}$ to the client indicating that it has reversed its transaction. If P_n already exists in L_s^{fail} the stock server resends the appropriate $M_s^{cancelled}$ to the client. If nothing exists in the database or L_s^{fail} the stock server records the P_n number as a failed purchase in its L_s^{fail} log using transaction T_s^{fail} and sends $M_s^{cancelled}$ to the client.

On receiving an $M_s^{cancelled}$ message the client sets F_c^{sfail} to TRUE using transaction T_c^{sfail}. As with F_c^{pfail}, this flag is originally set to FALSE. Once a client has both F_c^{sfail} and F_c^{pfail} set to TRUE the flexible transaction is considered failed. Once these two flags are set to TRUE a transaction is started at the client T_c^{fail} that deletes the original purchase order associated to T_c^{order} and resets all client flags.

3.3.2 Stock Server. Figure 4 provides an overview of what occurs when compensation is required due to the inability of the stock server to honor its obligations. Problems may occur at the stock server when attempting $T_s^{purchase}$ when the initial request for a purchase is made by a client (receives $M_c^{purchase}$). If this is the case then the stock server first checks its local database to determine if this message has been carried out previously in a successful manner (if P_n has previously been recorded by $T_s^{purchase}$). If P_n has already been processed then the stock server assumes that P_n is valid and may proceed as described without compensation. If P_n does not exist in the local database or the L_s^{fail} log then the stock server carries out a transaction T_s^{fail} to record P_n in the stock server's L_s^{fail} log. Once T_s^{fail} succeeds an M_s^{fail} message is sent to the client indicating that the purchase associated to P_n is not possible.

The storing of P_n in L_s^{fail} is for the same reason the payment server stored P_n in L_p^{fail} (to ensure the processing of a resent $M_c^{purchase}$ associated to a prior

failed purchase is not undertaken at a later time). Therefore, as in the payment server, all subsequent $M_c^{purchase}$ messages must only proceed if there is no prior record of them having failed in the stock server's L_s^{fail} log. If a $M_c^{purchase}$ message is received and an associated P_n number is already in L_s^{fail} then the stock server returns the previously generated M_s^{fail} message.

On receiving M_s^{fail} the client sets the F_c^{sfail} flag to TRUE using a transaction T_c^{sfail} and needs to now inform the payment server that any transaction that may have occurred to debit a customer's account needs to be reversed. Therefore, as long as F_c^{pfail} flag remains FALSE and F_c^{sfail} remains TRUE, the client must attempt to inform the payment server that the purchase associated to P_n has failed via a M_c^{fail} message.

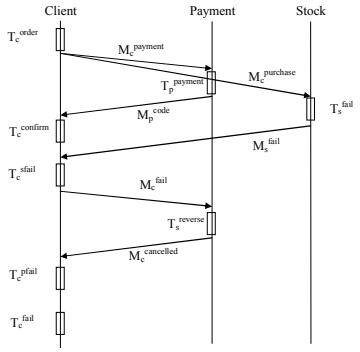

Figure 4 – Compensating in the presence of stock problems

On receiving M_c^{fail} the payment server determines if the transaction has already been compensated for by checking the database for failed prior compensated purchases within L_p^{fail}. If such a record already exists then an appropriate $M_p^{cancelled}$ message is sent to the client. If P_n is not recorded as failed or successful at the payment server (no record exists at all) then a transaction T_p^{fail} is used to record the failure of the purchase in L_p^{fail} and $M_p^{cancelled}$ is sent to the client. If P_n is recorded as a valid purchase then a transaction $T_p^{reverse}$ that reverses the associated payment in the database and records the failure in L_p^{fail} is attempted. Once $T_p^{reverse}$ has completed the payment server sends an $M_p^{cancelled}$ message to the client. On receiving $M_p^{cancelled}$ the client sets the F_c^{pfail} flag to TRUE using transaction T_c^{pfail}. The setting of both F_c^{pfail} and F_c^{sfail} flags to TRUE indicates that the purchase has failed and a transaction is started at the client T_c^{fail} that deletes the original purchase order associated to T_c^{order} and resets all client flags.

4. Conclusions

We have provided an overview of our initial work in implementing rich client based transactional systems. Focus of the paper is very much on the transactional process itself and we acknowledge that a complete solution will require a number of other issues to be addressed (e.g., security, catalogue caching, user identification). In addition, we acknowledge that our assumptions may be restrictive in that complete recovery from failure is expected from all participants to complete the overall process. Further work is required in this respect to identify how existing fault-tolerant techniques could be utilized to ensure progression in the presence of unrecoverable failure.

As a contribution, we have described how flexible transactions may be implemented on rich client platforms to implement a design pattern associated to the shopping cart. As such, empowering the client side with transactional capabilities provides additional possibilities for the developer with respect to existing tried and tested technologies.

5. References

[1] C. Kazoun, and J. Lott, 2008 "Programming Flex 3: the Comprehensive Guide to Creating Rich Internet Applications with Adobe Flex. 1", Adobe Dev Library - Imprint of: O'Reilly Media.

[2] Google, Google Desktop Website, as viewed November 2008, http://code.google.com/apis/desktop/

[3] Ebay, Ebay Desktop, as viewed November 2008, http://desktop.ebay.com/

[4] Amazon, Amazon Services Order Notifier (ASON), as viewed November 2008, http://www.amazon.com

[5] Adobe, Adobe Air, Website, as viewed November 2008, http://www.adobe.com/products/air/

[6] S. Frolund and R. Guerraoui, "e-transactions: End-to-end reli-ability for three-tier architectures", IEEE Transactions on Soft-ware Engineering 28(4): 378 - 395, April 2002

[7] J. Gray, 1981. The transaction concept: virtues and limitations (invited paper). In Proc. of the Seventh international Conference on Very Large Data Bases, Volume 7, pp 144-154, France, 1981

[8] G. Alonso, D. Agrawal, A. E. Abbadi, Kamath, M. R. Günthör, and C. Mohan, Advanced Transaction Models in Workflow Contexts. In Proc. of the Twelfth international Conference on Data Engineering, pp 574-58, Washington DC, 1996

[9] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres, Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In Proc.s of the 1994 ACM SIGMOD international Conference on Management of Data, Minneapolis, 67-78, 1994