Implementing Business Conversations with Consistency Guarantees using Message-oriented Middleware

Carlos Molina-Jimenez, Santosh Shrivastava and Nick Cook
School of Computing Science, Newcastle University
{Carlos.Molina, Santosh.Shrivastava, Nick.Cook}@ncl.ac.uk

Abstract

The paper considers distributed applications where interactions between constituent services take place via messages in an asynchronous environment with unpredictable communication and processing delays; further, interacting parties are not required to be on-line at the same time. Message-oriented middleware (MoM) is commonly used for connecting such loosely coupled distributed applications. Despite loose coupling, many service interactions have temporal and message validation constraints. A failure to deliver a valid message within its time constraint could cause mutually conflicting views of an interaction (one party regarding it as timely whilst the other party regarding it as untimely) leading to application level inconsistencies. In a loosely coupled system, such inconsistencies could remain undetected for a long time, requiring costly application level recovery procedures. This paper describes how synchronisation support providing multilateral consistency guarantees can be provided using the underlying MoM to prevent inconsistencies from reaching application level.

1. Introduction

Internet-scale distributed applications are increasingly being constructed by composing them from services provided by various businesses. A large class of such applications involves interacting parties that collaboratively execute a shared activity in a peer-to-peer relationship as against the more traditional client-server relationship. Interactions take place in a loosely coupled manner (interacting parties are not required to be on-line at the same time) in an asynchronous environment with unpredictable communication and processing delays. Message-oriented middleware (MoM) is commonly used for connecting such loosely coupled distributed applications.

We emphasize that in peer-to-peer relationships any peer can initiate the transfer of a message; messages are not necessarily paired as request-response. More importantly, regardless of the message flow, each peer exercises equal control on the status of the activity in the sense that, each peer can locally and unilaterally decide (at any time) on the correctness of a received message and on the final outcome of the interaction.

Unfortunately, loose coupling between the peers makes the task of maintaining consistency within the application very hard. Ideally, we would like that despite loose coupling, each peer has a mutually consistent view of the state of the application not withstanding software, hardware and network related problems (e.g., clock skews, unpredictable transmission delays, message loss, incorrect messages, node crashes etc.).

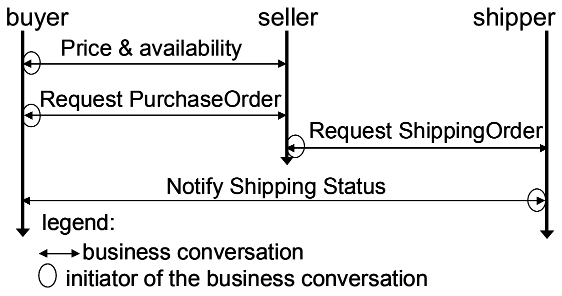


Fig. 1. B2B message interactions

We consider a simple example to set the scene. Fig.1 depicts interactions between three peers (buyer, seller and shipper) within a business-to-business (B2B) application concerned with purchase of goods. The notation used is: a double arrowed line indicates a *business conversation* (exchange of one or more electronic business documents) concerned with a well defined business activity (e.g., RequestPurchaseOrder, NotifyShippingStatus, and so on); the initiator of a conversation is identified by a circle on the arrow. The buyer first enquires with the seller about the price and availability of goods. If the required goods are available at acceptable price, the buyer initiates

RequestPurchaseOrder conversation with the seller. If this conversation is successful, the seller arranges shipping details with the shipper, who then informs the buyer of shipping details. This application will run well provided each conversation terminates with both sides having an identical view of the status of the conversation.

Business conversations usually have various Quality of Service (QoS) constraints (timing, security, message validation, etc.). RosettaNet partner interface processes (PIPs) [1,2], ebXML [3] and BizTalk [4] are examples of such specifications. They all have several timing and message validity constraints that need to be satisfied for successful completion (e.g., the sender of a notification of invoice message must receive its acknowledgement within 2 hrs). A failure to deliver a valid message within its time constraint could cause mutually conflicting views of an interaction (one party regarding it as timely whilst the other party regarding it as untimely). A conflict can also arise if a sent message is delivered but not taken up for processing due to some message validity condition not being met at the receiver (the sender assumes that the message is being processed whereas the receiver's validator has rejected it).

In a loosely coupled system, it could take a long time before such inconsistencies are detected. Subsequent recovery actions - frequently requiring application level compensation - may turn out to be quite costly. For example, assume that the buyer and seller have conflicting views about the outcome of a RequestPurchaseOrder conversation: the buyer regards it as successful (so is expecting goods to be delivered), whereas the seller has failed to complete the conversation successfully, therefore does not arrange goods delivery. The buyer could wait for a long time (several days perhaps) before suspecting something is amiss. The situation could be even worse if the misunderstanding is the other way round, so the goods are delivered to the buyer who is not expecting them. Existing B2B system architectures do not incorporate any specific solutions for preventing such inconsistencies from appearing at the application level. This paper remedies the situation by developing an architecture with synchronisation mechanisms that can be used to ensure that a given business conversation with QoS constraints terminates with both sides having an identical view of the status of the conversation. A simple and practical solution is presented. It supports the periodic synchronisation of the shared view of the interaction state, while preserving the desirable property of loose-coupling. We use RosettaNet PIPs to illustrate our ideas. However, our solution can be applied to arbitrary B2B interactions.

The paper is structured as follows: section two presents a layered architecture with synchronisation mechanisms for business conversations; section three describes the specific case of B2B interactions using RosettaNet PIPs; an implementation of our synchronisation service using Java Messaging Service (JMS) is presented in section four and related work is presented in section five.

2. Layered architecture

Three principal layers of the architecture are depicted in Fig. 2 for two partners (peers), a buyer and a seller.

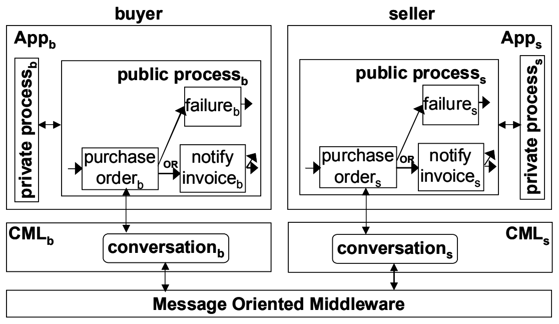


Fig. 2. Layered architecture

We use the *b* and *s* sub-index to denote buyer and seller, respectively. The business applications are implemented at the application layer (App$_b$ and App$_s$); each such application consists of a private and public process. Private process$_b$ and private process$_s$ represent, respectively, the core of the buyer's and seller's applications and are not directly related to each other; whereas public process$_b$ and public process$_s$ are complementary processes in the sense that they mirror each other. This complementary pair is also known as a *cross-organisational business process* or *shared business process* and can be regarded as a global business process executed between the two business partners.

The figure shows a segment of a public business process represented in a workflow style notation. If the 'purchase order' activity (the message interaction part of which is implemented by the lower level as a conversation) completes successfully, the buyer and seller move to execute the 'notify invoice' activity, else they move to execute the 'failure' activity and so forth. We want to coordinate the flow (runtime behaviour) of the shared process to guarantee that a given pair of complimentary activities always progress onto another pair of complimentary activities; in particular, we wish to avoid application level inconsistent situations such as the buyer side regarding the execution of purchase $order_b$ as successful and moving to execute notify $invoice_b$, whereas the seller side regarding the execution of purchase $order_s$ as failed and moving to execute $failure_s$.

The Conversation Management Layer (CML) ensures interoperability by implementing standards compliant conversations; for example, if the buyer and seller were to use the RosettaNet standard, this layer would contain the code that implements PIPs like, 3A4 for placing purchase orders, 3C3 for invoicing, and so forth. The arrows that link the business activities to the conversations represent the operations to transfer business data between the public business process and the CML layers.

The Message Oriented Middleware (MoM) layer is a messaging service that offers reliable, persistent message delivery with well-defined operations (e.g., connect, disconnect, transactionally deposit or pickup a message and so forth). In practice, the MoM could be a federation of several message queuing products, where messages are transactionally enqueued and dequeued, but such details need not concern us here.

One could envisage enclosing the execution of a business activity between the two partners within an atomic, 'ACID' transaction[1] thereby ensuring that both sides will have an identical view about the status of the conversation. Given that a business activity usually lasts a long time (e.g., several hours), this option is considered inappropriate[2]. Further, such a transaction spanning partners would also compromise their autonomy. In our architecture, any use of atomic transactions is confined to short-lived activities used locally by a partner for enqueuing/dequeuing messages, thus their use does not compromise the autonomy of the partners, in the spirit of loose coupling. Use of transactions is discussed further, in section four of the paper.

We classify failures in a system as (i) *temporary*: network partitions are assumed to heal eventually, so communication becomes possible, and participating nodes may crash but they will eventually recover; and (ii) *permanent*: partitions do not heal and participating nodes do not recover if they crash. Consistency requirements can not be guaranteed in the presence of permanent failures if the deployed consistency mechanisms use the same infrastructure components as the three layers; the best that can be achieved is maintaining consistency in the presence of temporary failures. This is what we will strive to achieve.

We assume that all the nodes have access to a common time service (such as the NIST Internet time service), which they use to keep their clocks synchronised to a reasonable (known and bounded) accuracy.

2.1. Consistency issues

At the application layers, business partners interact with each other by means of sending each other electronic documents. These documents are sent and received by means of invoking business operations that are intercepted by the public business process components. For example, the private $process_b$ invokes a "place purchase order request" operation when the buyer needs to send the seller a document requesting the purchase of some item; later on, perhaps upon receiving a notification, the private $process_b$ executes a "get purchase order response" operation to collect the response (another document) from the public $process_b$. The execution of these operations, results in the execution of complimentary business activities within the buyer's and seller's public processes.

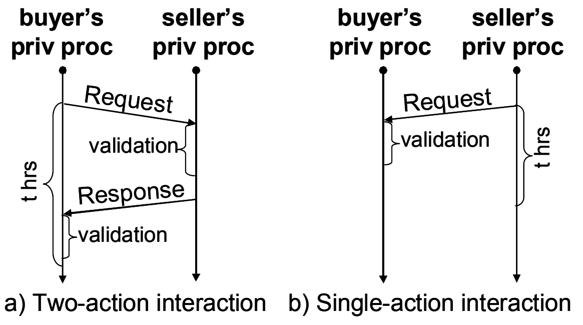


Fig. 3. Business interactions

As in RosettaNet, we consider two kinds of interactions with timing constraints: *single-action* and *two-action* (see Fig. 3, where priv proc stands for private process): in a single-action interaction only a single document is exchanged, whereas a two-action interaction involves exchange of two documents: a request and its response. The interactions shown in the figure depict what happens logically; in reality, each such interaction maps onto the corresponding conversation involving several messages implemented within the CML.

A received document (message) is accepted for processing by the receiver only if the document is received within the set timeout period (if applicable)

[1] ACID: Atomicity, Consistency, Isolation, Durability.

[2] It has long been realised that atomic transactions are not adequate for structuring long-lived applications [5,6].

and the document is *valid*. There are two validity checks that must be met:

1. Base-validation: the document must be *syntactically* valid; this involves verification of a static set of syntactical and data validation rules, according to the specification laid down in the standard being used; and in addition,
2. Content-validation: a base-validated document must also be *semantically* valid: document contents should satisfy some arbitrary, application specific correctness criteria.

The execution of a given activity can terminate either in a successful state (successful outcome) or in a failed state (failed outcome); the meaning is defined below for the two types of interactions. Our consistency requirement is that both the participants eventually reach the same outcome decision despite the occurrence of a finite number of temporary failures.

- Successful outcome of a two-action interaction: the request and response messages were valid and the response was received within the timeout period.
- Failed outcome of a two-action interaction: negation of the above.
- Successful outcome of a single-action interaction: the request message was delivered within the timeout period and the message was valid.
- Failed outcome of a single-action interaction: negation of the above.

It is clear that to reach an identical outcome decision, some additional interaction between the two business partners needs to take place after transmitting the Request (single action) and Response (two-action) messages shown in Fig. 3. For example, in a two-action interaction the seller would like to know whether its response message was considered timely and valid by the buyer. Ideally, the CML layer should provide the decision making facility. Both the timeliness and base-validation checks can be carried out within the CML layer; however, content-validation will necessarily involve application level processing. Unfortunately, this complicates matters, which we resolve by proposing a two level synchronisation mechanism that will be discussed in the next subsection.

The conversation that implements a two or one action interaction will implement a protocol that will involve the exchange of several messages (transmissions, retransmissions, acknowledgements, etc.) under strict time and base-validation constraints.

An example of a conversation that is meant to realise the execution of a single action interaction is shown in Fig. 4. The example assumes that the CML layers are directly using the Internet transport service for implementing the conversation. We assume that the specification of the conversation states that an acknowledgement should be received within *t hrs* after sending a request message.

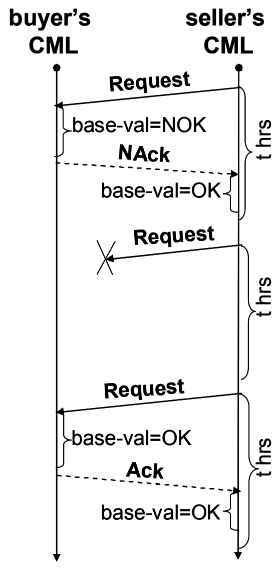


Fig. 4. Example of a conversation

Notice that all messages are base-validated. We assume that Ack and NAck messages are generated automatically and need only base-validation. A NAck message is sent to the seller in response to a Request message that fails its base-validation; assuming that retransmissions are allowed, this failure results in the retransmission of the Request message; unfortunately, it is lost in the network (arrives too late); so after failing to receive an Ack within *t hrs*, the seller retransmits the Request message, this time it is successfully base-validated and acknowledged by an Ack message. This Ack messages arrives on time and is successfully validated. An issue that emerges from this discussion is the possibility that the last Ack fails to arrive on time or to meet base-validation requirements. The problem here is that the buyer has no means to know the fate of its last Ack. This is the well known last Ack problem in protocol design [7]. The buyer has no choice but to unilaterally declare its conversation successfully completed and hope for the best. Most of the times, the buyer's guess will be correct, however, once in a while, the buyer and the seller will have conflicting views over the completion of the conversation, whose resolution can be performed using a communication synchronisation service as we discuss next.

2.2. Synchronisation

From the previous discussion it should be clear that execution of a public process results is complex interactions that could drive the participants into misunderstandings unless the designer envision mechanisms to synchronise them. Since timeliness and base-validation checks can be performed automatically at the level of CML, it is best to have a mechanism for reaching agreement on such checks and to have a separate mechanism to handle semantic errors that emerge from the content validation of Response messages (two-action interaction) and from Request messages in single-action interactions. Notice that the Request message of a two-action interaction might contain semantics errors too; however, this problem can be corrected within the conversation with the help of negative acknowledgements. Semantic checks could involve arbitrary application level processing, so could be time consuming. We propose two synchronisation mechanisms: communication synchronisation and application synchronisation (see Fig. 5) that can be combined in a modular fashion.

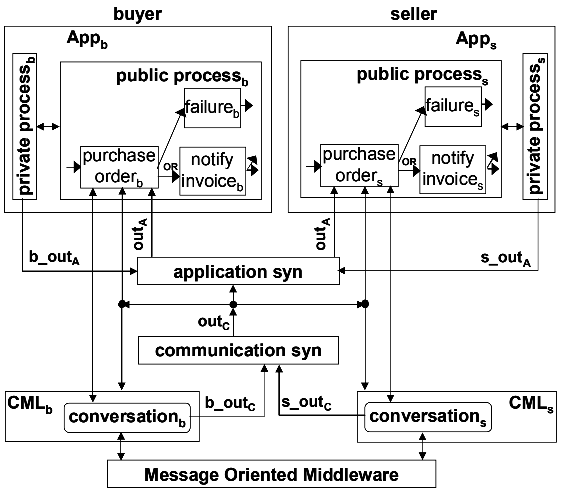


Fig. 5. Synchronisation mechanisms

Communication synchronisation: The job of the communication syn is to detect conflicts emerging at the CML layer, that is, conflicts due to a lost, delayed or base-invalid last Ack message. The central idea is that once one of the business partners considers the conversation completed it starts the execution of a synchronisation protocol with its counterpart to agree on a single conversational outcome, out_C which equals either *success* or *failure*. The out_C arrow splits into several communication channels to suggest that out_C can be presented to the application syn, CML$_b$, CLM$_s$, purchase order$_b$ and purchase order$_s$. The local outcomes of conversation$_b$ and conversation$_s$ are represented as b_out_C and s_out_C, respectively. A successful out_C means that the document contained in the last message can be passed up by conversation$_b$ and conversation$_s$ to the respective public business processes. If a out_C=*failure* is agreed, it can be presented to the CML layer where the conversation can be re-started or the error can be notified to purchase order$_b$ and purchase order$_s$ which will progress in harmony to failure$_b$ and failure$_s$, respectively, perhaps, after notifying their private processes of the problem.

In summary, successful outcome from communication synchronisation provides the following (*weak*) consistencies guarantees for a given interaction:

- Two-action interaction: request message was valid and the response was received within the timeout period and was base-valid.
- Single-action interaction: the request message was delivered within the timeout period and the message was base-valid.

Application synchronisation: Application synchronisation of an interaction is performed after receiving a successful outcome (out_C=*success)* from communication synchronisation. It provides the strong consistency guarantees described in the previous sub-section. Its job is to synchronise the two business partners when conflicts derived from semantic validation emerge. In the figure, the b_out_A and s_out_A messages fed into the application syn represent the buyer's and seller's local and possibly conflicting outcomes of the execution of the business activity; out_A represent an agreed upon outcome provided by the application syn mechanism. In the case of the two-action interaction (Fig. 3), b_out_A is produced by the buyer upon content-validating the response message; s_out_A represents the seller's view of the outcome of the activity and is always *success*, since at this stage the seller has already successfully met his time and validation constraints. In fact, the value of s_out_A is assigned from out_C.

The deployment of the application level synchronisation has two significant implications.

1. It ensures that public processes always deal with valid documents. This means that there is no need for error recovery (typically requiring compensation) caused by processing of 'dirty data': the situation that lead to one side processing a document that the other side might still reject as invalid.
2. A consequence is that it forces the two business partners to progress their local execution of public processes in a lock-step mode: the faster business partner has to wait until the slower one completes.

If this is considered undesirable, then a possible compromise solution would be for the partners to use just the communication synchronisation and accept the risk of using dirty data.

Preliminary ideas on the need for synchronisation for business processes were developed in our paper [8], where we sketched out distributed as well as centralised synchronisation approaches. A distributed synchronisation mechanism is described in [9]. A centralised solution that can be implemented using a MoM is described in this paper, in section four. It can be used for both communication and application synchronisation.

3. RosettaNet conversations

In this section we discuss how business activities can be mapped into RosettaNet Partner Interface Processes (PIPs) and how the issues we discussed in previous sections manifest. RosettaNet is a globally supported B2B standards organisation that has published several specifications known as PIPs that define basic business conversations [1,2].

3.1. Time and validation constraints

A good example of a RosettaNet PIP is the purchase order PIP 3A4 which is expected to be used by a buyer to express his desire to place a purchase order with a seller. On the order hand, the seller is expected to use the Notification of Invoice, PIP 3C3 to invoice the buyer. RosettaNet distinguishes between *business action* and *signal* messages. Business action messages (referred to from now on as action messages) are sent to indicate a business activity such as a purchase order request, purchase order confirmation, invoice notification etc. Signal messages (also called acknowledgements) are messages sent in response to action messages.

PIPs in RosettaNet fall into two categories: single-action and two-action activity. In single-action activity PIPs, only one action message is sent from the initiator to the responder followed by an acknowledgement sent back to the initiator. In two-action activity PIPs, two action messages are sent (with their respective acknowledgements); one (request) from the initiator to the responder and one (response) from the responder to the initiator.

A graphical representation of PIPs 3A4 (a two-action PIP) and 3C3 (single-action PIP) is shown in Fig. 6 where action and signal messages are represented by solid and dashed lines, respectively. PIP messages must pass base and content validation tests before being accepted for processing, as explained in the previous section. Similarly, PIP messages have time constraints. In a PIP, the delivery time is explicitly defined in the message choreography.

For example, the sender of PurchaseOrderRequest action message expects to receive a base-valid acknowledgment (termed a positive acknowledgment or Ack) within 2 hrs; likewise, the initiator of a two-action activity PIP expects to receive a base-valid response (an action message) within 24 hrs. Timing, base-validation and content-validation exceptions can drive the two PIP executors out of synchrony.

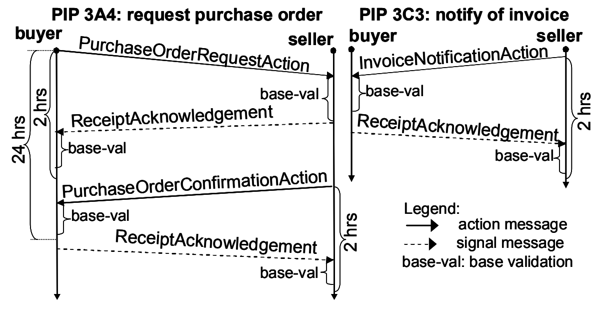
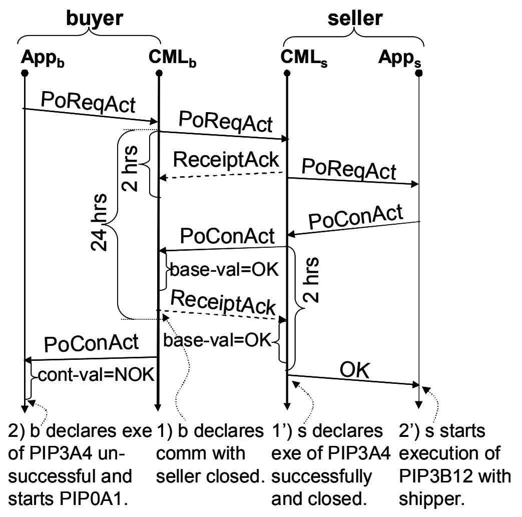


Fig. 6. RosettaNet PIP3A4 and PIP3C3

RosettaNet tries to remedy these situations at PIP execution level by means of negative acknowledgements (NAck) and retransmissions [10]. As shown in Fig. 4, in RosettaNet a base-invalid action message is acknowledged by a NAck to encourage the sender to retransmit its action message; similarly, if nothing is received within the 2 hrs frame, the action message is retransmitted. In two-action PIPs the response (second action message) is sent only when the request (first action message) satisfies its content validation test; a NAck (that overwrites a previously sent Ack) is sent to trigger the retransmission of the request, when the content-validation test fails. Up to three retransmissions are allowed. Base-invalid Acks and NAcks are treated as not received. Notice that with two-action PIPs the initiator unilaterally closes the communication with the responder, immediately after sending the last Ack regardless of the Ack being base-valid or received on time. On the other side, the responder unilaterally closes the communication when it receives a base-valid Ack. Likewise, with single action PIPs, the responder unilaterally closes the communication immediately after sending the Ack regardless of the Ack being base-valid or received on time; on the other side, the initiator unilaterally closes the communication immediately after receiving a base-valid Ack.

3.2. Notification of failure

A question that arises from the previous discussion is how to handle situations where the last Ack (in single and two-action PIPs) is either not received on time or is base-invalid. Similarly, how to handle situations where: (i) in a two-action PIP, the initiator's application finds the second action message content-invalid; and (ii) in a single-action PIP, the responder's application discovers that the action message is content-invalid. The particularity of these situations is that the offending business partner has already closed the communication under the assumption that everything is normal; consequently, the strategy of sending NAcks to trigger retransmissions at PIP level cannot be used. In RosettaNet, the suggested solution is to address the problem at the application level where the party that detects the error instigates a special PIP for the notification of failure conversation (PIP 01A). PIP 01A execution is normally done though an alternative out of band channel. To illustrate this possible solution we will refer to the scenario shown in Fig. 1. The flow of messages is illustrated in Fig. 7.



buyer
seller
App$_b$
CML$_b$
CML$_s$
App$_s$
PoReqAct
PoReqAct
2 hrs
ReceiptAck
PoReqAct
24 hrs
PoConAct
PoConAct
base-val=OK
2 hrs
ReceiptAck
PoConAct
base-val=OK
cont-val=NOK
OK
2) b declares exe of PIP3A4 un-successful and starts PIP0A1.
1) b declares comm with seller closed.
1') s declares exe of PIP3A4 successfully and closed.
2') s starts execution of PIP3B12 with shipper.

Fig. 7. Delivery of unexpected goods

In the figure, as before, *b* and *s* stand for buyer and seller respectively, whereas *exe* and *comm.* stand for execution and communication, respectively. Further, *PoReqAct*, *ReceiptAck* and *PoConAct* stand for purchase order request action, receipt acknowledgement and purchase order confirmation action, respectively. We assume that PoReqAct and its ReceiptAck are successfully received and validated. An ideal execution of PIP 3A4 requires that PoConAct is successfully content-validated by the buyer and that its ReceiptAck arrives on time and is successfully base-validated by the seller. Now imagine for example that the last ReceiptAck is received on time and successfully base-validated by the seller, so the latter instigates the execution of PIP 3B12 with the shipper. Unfortunately, on the buyer's side the application finds the message PoConAct to be content-invalid; so the buyer starts the execution of notification of failure PIP 01A with the seller to signal the situation and ask for synchronisation. As observed earlier, in a loosely coupled system, it could take a long time before such inconsistencies are detected. For example, in our scenario, the shipper might already be knocking on the buyer's door with the goods by the time the seller receives the PIP 01A signal.

4. Synchronisation service implementation

Conceptually speaking, the job of the two synchronisation mechanisms (communication syn and application syn) shown in Fig. 5 is to coordinate the progress of the two business partners. The actual implementation of these mechanisms in a MoM setting is described in this section; before that some observations on the use of transactions are in order.

4.1 Transactional execution of conversations

We stated earlier that we rule out the solution that encloses a conversation into a single atomic transaction. We expand on this observation here. An atomic transaction will require close-coupling in the sense that the two partners will need to be on line during the whole execution of the conversation which might last several hours; during this execution time the execution autonomy of the buyer and seller is seriously compromised as both of them have to grant control of their local resources to the transaction controller who normally locks them until the transaction commits or aborts. Matters can get worse when failures occur during commit processing: the unavailability of one of the parties blocks the progress of the other; likewise a crash of the coordinator would lock the buyer's and seller's resources until the coordinator recovers and commits or aborts the transaction. In practice therefore, transactions spanning autonomous entities are rarely used [11].

The preferred approach is to use queued transaction processing (QTP) paradigm, introduced in [12], and further discussed in [13] that works under the principle of unilateral commit [14]. To send a request to a server, a client starts a transaction, deposits the request into a client-to-server queue and commits. Then, the server starts a transaction, retrieves the request from the client-to-server queue, processes it, deposits any reply in a server-to-client queue and commits. Finally, the client starts a transaction, retrieves the reply from the server-to-client queue, processes it and commits.

In Fig. 8 we show the implementation of the single-action PIP 3C3 in which buyer and seller communicate by the transactional deposit and retrieval of messages from MoM-maintained persistent queues. The seller initiates transaction T_{s1} to deposit a Notify of Invoice Action message (InvNotif) into the MoM's queue. Base and content validation (base-val and cont-val, respectively) have to be performed at some point.

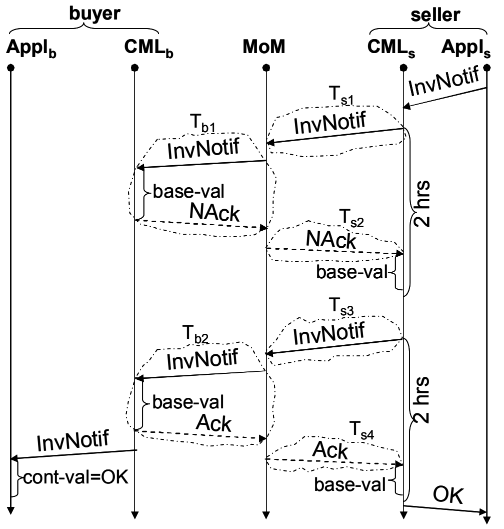


Fig. 8. Execution of a single-action PIP in a QTP environment

In the example, the buyer initiates the T_{b1} transaction to remove the InvNotif message from the MoM, next it base-validates the message; in this scenario, base-validation fails, consequently, the buyer deposits a NAck into the MoM and commits T_{b1}.The seller initiates T_{s2} to retrieve its acknowledgement (a NAck in this case) and commits. The NAck encourages the seller to initiate transaction T_{s3} to retransmit the InvNotif message; once the message is deposit into the MoM, T_{s3} commits. The buyer initiates transaction T_{b2} to retrieve the InvNotif message from the MoM; it successfully base-validates it, deposits an Ack into the MoM and commits T_{b2}. The seller initiates the T_{s4} transaction to retrieve the Ack and commits T_{s4}. Whether to perform validation inside or outside of message dequeuing transactions is a matter of design choice; we have arbitrarily decided to base-validate acknowledgements outside the receiving transactions; this is why NAck and Ack are base-validated (successfully in this case) outside T_{s2} and T_{s4} respectively. Likewise, content-validation is performed outside the buyer's T_{b2} transaction; again, in our scenario InvNotif is content-valid.

Whatever approach is adopted, transactions T_{b2} and T_{s4} only provide unilateral local views of the outcome. The use of QTP alone cannot guarantee a consistent, multi-lateral view of the outcome of a conversation at the CML layer.

4.2. Implementation

In this section we describe how a synchronisation mechanism like the two shown in Fig. 5 can be implemented easily using the publish/subscribe facility of a MoM. We will describe a generic synchronisation service (see Fig. 9) under the understanding that it can be used for both or any of the two.

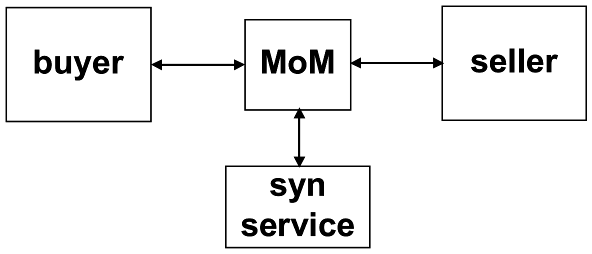


Fig. 9. Synchronisation service

As suggested by the figure, the corner stone of our solution is a conventional MoM which works as a messaging server and to which a buyer, seller and our synchronisation service (SynService) subscribe as clients. In our prototype, we use an implementation of the Java Message Service (JMS) [15].

Basically, conversation consistency is achieved by means of publish/subscribe JMS topics. Conversation participants (the buyer and seller in the example) publish messages to a synchronisation topic to which the SynService subscribes. The messages uniquely identify the business conversation (PIP instance) to which they relate. The sequence of activities is straightforward:

1) To initiate the synchronisation of a conversation both, the buyer and seller register with the MoM as publishers to the synchronisation topic. The SynService registers as a subscriber.

2) When the buyer and seller complete their executions (not necessarily at the same time), they send their individual outcomes to the MoM to be published under the synchronisation topic.

3) Eventually, the two outcomes will be delivered to the SynService. In addition to the Boolean outcome (*success* or *failure*) and conversation identifier, the messages produced by the buyer and seller can contain

other relevant information to help the SynService to compute and deliver the global outcome. For example, the messages may identify the sender, the name of other senders that are expected to provide individual outcomes, the timeout to produce the global outcome and the JMS destinations to which the global outcome should be delivered.

4) The result of the synchronisation is delivered to the interested parties by means of the publish/subscribe mechanism where the SynService plays the role of the publisher to JMS destinations and the buyer, seller and other interested parties (e.g., a shipping company) are subscribers.

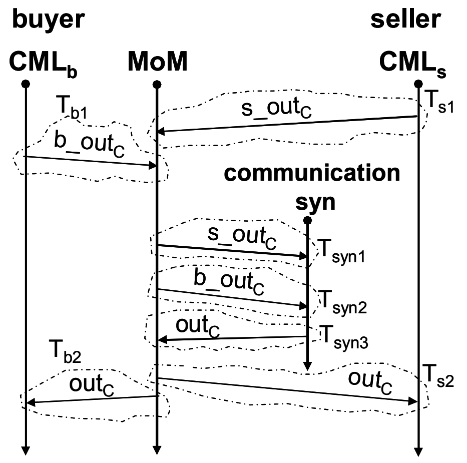


Fig. 10. Communication synchronisation of a single-action PIP

A sequence diagram of communication synchronisation of a single-action conversation is shown in Fig. 10. Notice that the queuing and dequeuing of messages are protected by transactions (t_{b1}, t_{b2}, t_{s1}, t_{s2}, T_{syn1}, etc.). Let us assume, for example, that in the figure, CML_b and CML_s have just completed the PIP 3C3 conversation shown in Fig 8. CML_b and CML_s present their individual outcomes to the MoM, b_out_C and s_out_C, respectively. The two outcomes are delivered to the communication syn that computes the global outcome (out_C) and sends it back to the MoM. Eventually, the out_C message is delivered to CML_b and CML_s.

Once a result message has been sent and a consistent view of conversation status has been set, the communication syn will respond with the same result message to any subsequent synchronisation request for the same conversation.

For simplicity, we assume that the global outcome is computed by the communication syn as a conventional AND function. A *failure* outcome from any of the partners immediately results in a global outcome equal to *failure*, whereas a *success* outcome will cause the communication syn to wait till the last individual outcomes arrives. It is sensible to consider that a trading partner's outcome never arrives. In this situation, the communication syn times out, assigns *failure* to the missing values, computes its AND function and publishes its global result. Outcome messages for synchronisation could be extended to specify more complex decision functions to compute conversation status.

5. Related work

The benefits of conversation-centric business interactions have been pointed out by several authors. In [16] for example, the authors discuss a business process integration model where conversations (sequence of message exchange with formatting, timing and validation constraints) are executed between loosely coupled business partners; however, the treatment of validation errors is not addressed.

Maintaining consistency is an issue that will arise in any distributed application that involves coordinated execution of activities between two or more parties. Business applications are not an exception. Our objective in this paper is to study inconsistencies that emerge at the communication layer and propagate to the application layer unless they are detected and addressed. To place our discussion in context of the overall system, it is worthwhile examining the layered architecture depicted in Fig. 2. In multi-level architectures, different kinds of inconsistencies (exceptional situations) can occur at each level and propagate up to the level above unless mechanisms to detect and mask them are deployed. The question that arises is whether the exceptions should be masked at a given level or allowed to propagate up to the next level. The end-to-end argument in system design [17] discusses the design issues involved.

One extreme possibility is to leave it to the application to detect and correct the consequences of all potential inconsistencies regardless of the level in which they occurred. A notable example of a commercial product where this approach is taken is Microsoft BizTalk Framework 2.0 [4]. BizTalk is a specification for B2B interactions over asynchronous messaging that specifies message validation and timing constraints. The sender can specify a deadline to receive an acknowledgement that the sent document has been received and accepted (deliveryReceipt) and a second acknowledgement (commitmentReceipt) indicating that the destination has received the document and that it is committed to process it. Likewise, it can specify a time period (expiresAt) beyond which a document, if unprocessed, becomes null and void. For example, a failure to receive a commitmentReceipt message before the deadline, results in an error sent by the offended BizTalk server to its application which is expected to remedy the situation. The difficulty here is that the offended application does not know what has happened at its counterpart's side: it is quite possible that the receiver refused to process the document or that it did not have enough time to process it. Also, as briefly mentioned in the specification, there is a small but finite possibility that the commitmentReceipt was sent but arrived too late; a situation like this would result in inconsistent views over the interaction.

The layered architecture of our approach contrasts with that presented in [18]. This work describes an implementation of RosettaNet PIPs at application level that assumes only limited underlying support for reliable message delivery and no support from the CML layer. Our approach can be seen as residing at the other end of the spectrum: the communication layer deals with problems of message loss and decoupled execution by providing persistent, queued messaging. Conversation level problems are handled at the CML layer and so on.

The use of MoM considerably simplifies the task of constructing business conversations because the MoM takes on the responsibilities of reliable message delivery between decoupled systems. For example, [19] describes how Web services could be enhanced with MoM.

At present, there are several vendors offering messaging products; unfortunately, they are not interoperable. The need for interoperable messaging middleware is widely recognised and there are efforts aimed at developing a standard [20].

The idea of introducing outcome synchronisation for business conversations was first introduced in [8] where we showed how PIP executions can be 'wrapped' by simple handshake protocols to provide bilateral consistency, thereby simplifying the task of coordinating peer-to-peer business processes. In [9] we show how synchronisation can be achieved by means of a three-way handshake protocol executed between the two business partners. The solution suggested is distributed and suitable for synchronising pairs of business partners that do not wish to deploy any third party synchronisation services like the one discussed here. The idea behind our synchronisation service can be seen as a specialisation of the general concept of a consensus service proposed by the authors in [21].

The potential use of publish/subscribe infrastructures enhanced with transactional facilities is discussed in [22]. The authors propose that event notification and its processing by one or more subscribers be enclosed within a transaction. As admitted by the authors, atomicity is guaranteed at the cost compromising loose coupling. With a single-action PIP like PIP 3C3, the transaction would need to enclose the sending of the InvNotif message, its base and content validation, and the sending and base-validation of the Ack message. The transaction will commit only if the content-validation of the InvNotif message and the base validation of the Ack message are satisfactory. In some way, this solution is similar to ours with application level synchronisation with the exception that ours does not compromise loose coupling simply because we let the parties to communicate asynchronously, process and declare results unilaterally and only then (and before the application sees the unilateral outcomes) we activate our sync service.

There has been much recent work on business transactions in Web services environment with relaxed atomicity properties, such as the proposed standards WS-Coordination, WS-Transactions [23]. They provide highly desirable application level mechanisms for arranging coordinated recovery and compensation. Our approach can be seen as providing complimentary consistency mechanisms intended mainly for controlling lower level message interactions.

6. Concluding remarks

A primitive B2B interaction normally results in a business conversation run to exchange one or two electronic business documents (e.g., a purchase order, shipping notification, an invoice) and has various QoS constraints (timing, security, message validation, etc.). Message exchange patterns of the type shown in Fig. 6 are good examples of such conversations. Although these specifically belong to the RosettaNet B2B standard, other standards, such as ebXML [3] make use of similar patterns. Thus it is worth investigating what middleware support can be provided. A MoM provides a good foundation for supporting such interactions as it can take on the responsibilities of reliable message delivery (e.g., exactly once) between decoupled systems. We described how a very useful form of synchronisation support can be incorporated.

The synchronisers discussed in this paper can provide a general-purpose service for agreement on a set of inputs from business partners. They require no knowledge of the domain to which the inputs relate. They simply need a specification of how to compute a consistent view over the inputs. The use of a MoM allows them to scale to support interactions with many more than just two partners. There is considerable flexibility in the user-specification of destinations for synchronisation results. This allows business partners to independently determine where results should be delivered (to queues and/or to topics). Indeed, it is possible for processes beyond the conversation participants, including applications at other organisations (e.g. a shipping company), to subscribe to named topics for notification of results. The synchronisers could also be extended to deliver fairness and security guarantees such as non-repudiation [24].

Acknowledgments

This work is funded in part by UK Engineering and Physical Sciences Research Council (EPSRC), Platform Grant No. EP/D037743/1, “Networked Computing in Inter-organisation Settings”

References

[1] “RosettaNet implementation framework: core specification, Version V02.00.01,” 6 Mar. 2002, http://rosettanet.org.

[2] S. Damodaran, “B2B Integration over the Internet with XML – RosettaNet Successes and Challenges”, *Proc. WWW2004 Conf.*, May 17-22, 2004, New York, USA, ACM pp. 188-195.

[3] A. Grangard, B. Eisenberg, D. Nickull, et al., “ebXML Technical Architecture Specification v1.0.4. OASIS Final Draft,” 2001, http://www.ebxml.org/.

[4] “BizTalk Framework 2.0: Document and Message Specification,” Dec.2000, http://www.microsoft.com/biztalk/.

[5] J.N. Gray, “The transaction concept: virtues and limitations”, *Proc. 7th VLDB Conf.*, Sep. 1981, pp. 144-154.

[6] D.J. Taylor, “How big can an atomic action be?,” *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, Jan. 1986, pp. 121-124.

[7] A. Tannenbaum, *Computer Networks*, Prentice Hall, ISBN: 0-13-066102-3, 2003.

[8] C. Molina-Jimenez, S. Shrivastava and S. Woodman, “On State Synchronisation of Business Conversations,” *Proc. IEEE Conf. on E-Commerce Technology (CEC'06)*, June 2006, San Francisco, pp. 324-327.

[9] C. Molina-Jimenez, S. Shrivastava, “Maintaining consistency between loosely coupled services in the presence of timing constraints and validation errors,” *Proc. 4th IEEE European Conf. on Web Services (ECOWS’06)*, Dec. 2006, Zurich, pp. 148-157.

[10] “RosettaNet implementation framework, Version V02.00.01: High Availability Features, technical Recommendation,” Dec. 2004.

[11] P. Helland, “Life beyond Distributed Transactions: an Apostate's Opinion,” *Proc.3rd Biennial Conf. on Innovative Data Systems Research (CIDR’07)*, Asilomar, CA, USA, 7-10 Jan., 2007.

[12] P.A. Bernstein, M. Hsu and B. Mann, “Implementing recoverable requests using queues,” *Proc. 1990 ACM SIGMOD Intl. Conf. on Management of Data,* Atlantic City, New Jersey, 1990.

[13] S. Tai and I. Rouvellou, “Strategies for Integrating Messaging and Distributed Object Transactions," *IFIP/ACM Int. Conf. on Distributed Systems Platforms and Open Distributed Processing,* 4-8 Apr., New York, 2000.

[14] M. Hsu and A. Silberschatz, “Unilateral commit: a new paradigm for reliable distributed transaction processing,” *Proc. 7th Intl Conf. on Data Engineering,* 8-12 Apr., Kobe Japan, 1991.

[15] “Sun Microsystems. Java Message Service Specification (JMS), Specification 1.1, Sun Microsystems Inc.,” 2002, http://java.sun.com/products/jms/.

[16] J.E. Hanson, P. Nandi and S. Kumaran, “Conversation Support for Business Process Integration,” *Proc. 6th Int’l. Enterprise Distributed Object Computing Conf. (EDOC'02),* Sep. 17-20, Ecole Polytechnic, Switzerland, 2002.

[17] J.H. Saltzer, D.P. Reed and D.D. Clark, “End-to-end argument in system design,” *ACM Transactions on Computer Systems,* vol.2, no.4, Nov. 1984, pp. 277-288.

[18] R. Khalaf, “From RosettaNet PIPs to BPEL Processes: A Three Level Approach for Business Protocols,” *Data & Knowledge Engineering,* Elsevier, no. 61, 2007, pp. 23–38.

[19] P. Maheshwari, H. Tang and R. Laing, “Enhancing Web Services with Message-Oriented Middleware,” *Proc. IEEE Int’l. Conf. on Web Services (ICWS’04),* July 2004, San Diego.

[20] “AMQP, Advanced Message Queuing Protocol Specification,” Dec. 2006, www.amqp.org.

[21] R. Guerraoui and A. Schiper, “Consensus Service: a modular approach for building agreement protocols in distributed systems,” *Proc. 26th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-26),* 25-27 June 1996, pp.168-177.

[22] L. Vargas, J. Bacon and K. Moody, “Transactions in Distributed Event-Based Middleware,” *Proc. 3rd Int’l. Conf. on Enterprise Computing, E-Commerce, and E-Services (EEE'06),* Palo Alto, USA, 26-29 June, 2006.

[23] G. Alonso, F. Casati, H. Kuno and V. Machiraju, *Web Services: Concepts, Architectures and Applications*, Springer-Verlag, 2003.

[24] N. Cook, P. Robinson and S. Shrivastava, "Design and Implementation of Web Services Middleware to Support Fair Non-repudiable Interactions," *Int. J. of Cooperative Information Systems (IJCIS), Special Issue on Enterprise Distributed Computing,* 15(4), Dec. 2006, pp. 565–597.