Communication Abstractions for Distributed Business Processes

Lachlan Aldred[1], Wil M.P. van der Aalst[1,2], Marlon Dumas[1], and Arthur H.M. ter Hofstede[1]

[1] BPM Group, Queensland University of Technology, Australia
{l.aldred,m.dumas,a.terhofstede}@qut.edu.au
[2] Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands
w.m.p.v.d.aalst@tue.nl

Abstract. Languages for business process definition generally suffer from myopic approaches to capturing communication between distributed processes. Effective communication between processes requires: support for conversations involving interrelated interactions spread over time; ability to select and group messages based on their content, regardless of format and transport technology; and resolving contention between processes or tasks for common sets of messages. This paper presents a set of communication abstractions that provide a "glue" between the process layer and the middleware. The paper also reports on an implementation of these abstractions and an experimental evaluation.
Keywords: Business process integration, correlation patterns.

1 Introduction

At present, the definition of business processes that interact with one another in a distributed environment is hampered by a number of factors. Firstly, these processes are required to run on top of mainstream communication middleware which often does not support key features needed by applications in general [4, 10, 12, 11], and process-oriented applications in particular. For instance message selectors in JMS[3] cannot filter messages based on their body. Secondly, there exist conceptual problems with state-of-the-art business process definition languages in regards to process-to-process communication abstractions. To receive message batches for example, Business Process Management Systems (BPMSs) force designers to incorporate dedicated code into the same scope as business process logic. This leads to brittle process definitions. Finally, apart from correlation mechanisms for routing messages to process instances, BPMSs accept messages sent to them without filtering, thus forcing unnecessary amounts of message selection code into the process definition. In summary, the distinction between process abstractions and communication abstractions is blurred in existing approaches.

[3] Java Message Service: java.sun.com/products/JMS accessed November 2006.

This paper motivates and defines a set of communication abstractions at a layer in-between the business process and the middleware, that simplify the definition of interactions between distributed processes. The proposed abstractions have been implemented on top of a communication API, namely JCoupling [5], that abstracts away from the underlying middleware and protocols. Using this implementation, we have conducted preliminary experiments to compare this approach with respect to the approach embodied in the business process execution languages for Web services (WS-BPEL) [7].

Figure 1 presents the scope of our proposal within the broader context of business process integration. At the time of writing there are efforts to enhance any layer mentioned. In practice, however, the top three layers (Contract, Discovery, and Choreography) are almost exclusively performed by people, and the bottom two layers (Messaging, and Transport) are performed by almost every form of middleware available.

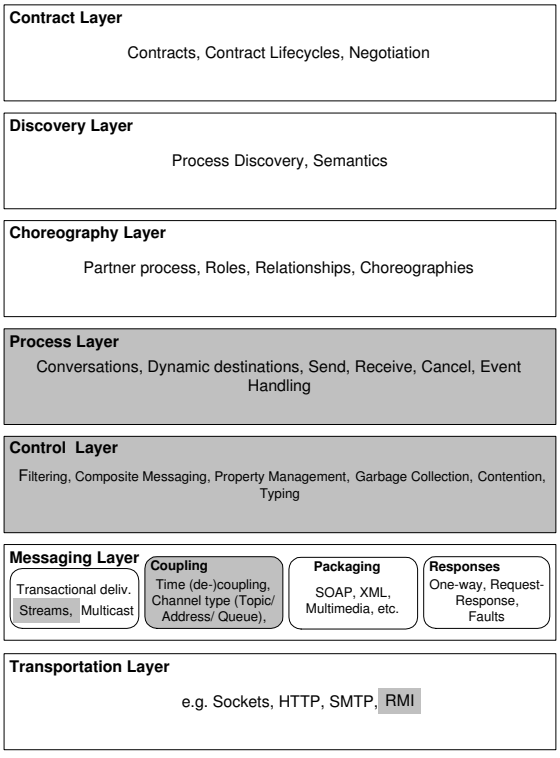

Fig. 1. Layers supporting process integration.

The remaining two layers (Process, and Control) are where the bulk of current effort is being directed. WS-BPEL, for example touches heavily on these two layers, and also deals with Roles and Relationships from the Choreography layer. These two layers also form the scope of this paper.

The next section presents motivating scenarios and requirements for process-to-process communication. Section 3 introduces a communication model for distributed business processes addressing these requirements. Section 4 discusses the implementation and experimental evaluation of a prototype that implements these abstractions. Section 5 discusses related work and Section 6 concludes.

2 Motivating Requirements

Implementing interactions between business processes brings new issues on top of the traditional requirements found in distributed systems implementation (e.g. coupling over time and space [4, 12], guaranteed delivery, encryption). In this section we distill some of these specialised requirements. These requirements are drawn from two studies on patterns in the area of integration of enterprise applications: the Enterprise Integration Patterns by Hohpe and Woolf [14], and the Correlation Patterns by Barros et. al. [9].

The correlation patterns described by Barros et. al. review conversation and message consumption patterns for distributed business processes. Most of those patterns led to requirements motivating our proposals. From Hohpe and Woolf we categorised the patterns into (1) Patterns forming requirements for a BPM messaging solution, (2) Patterns that are supported by most middleware, (3) Patterns composable from any data-aware, message-aware system, (4) Patterns concerning deployment and administration. The communication abstractions proposed in this paper focus on the first category. Following is our categorisations of Hohpe and Woolf's patterns.

Patterns that constitute requirements for a BPM messaging solution
Aggregator, Channel Purger, Competing Consumers, Correlation Identifier, Data Type Channel, Event Driven Consumer, Message Expiration, Message Filter, Polling Consumer, Publish-Subscribe Channel, Remote Procedure Invocation, Request-Reply, Return Address, Scatter-Gather, Selective Consumer, Transactional Client

Well understood patterns that are supported by any middleware solution
Command Message, Document Message, Event Message, Guaranteed delivery, Message Endpoint, Message, Messaging, Point to Point Channel, Message Channel

Patterns composable from any data aware, message aware system
Canonical Data Model, Claim Check, Composed Message Processor, Channel Adapter, Content Based Router, Content Enricher, Content Filter, Detour, Dynamic Router, Envelope Wrapper, Format Indicator, Idempotent Receiver, Invalid Message Channel, Message Broker, Message Bus, Message Dispatcher, Message History, Message Router, Message Sequence, Message Store, Message Translator, Messaging Bridge, Messaging Gateway, Messaging Mapper, Normalizer, Pipes and Filters, Process Manager, Recipient List, Resequencer, Routing Slip, Service Activator, Shared Database, Smart Proxy, Splitter, Test Message, Wire Tap

Patterns concerning deployment/administration Control Bus, File transfer

In the remainder of this section, we discuss the different requirements. For each requirement we cite the related patterns from [9, 14].

Conversations Support for conversations is necessary when business processes need to exchange more than one related message, and in particular when the processes are stateful, or execute over long periods. Furthermore typical process deployments, require many instances to share common channels and conversations enable messages finding their way to the correct instance. Proposals such as WS-BPEL may often create the illusion that there is one process instance, when in fact there are many – because all process instances are hidden behind the one portType/channel. We know of four approaches to implementing conversations through correlation:

1. *Property-based Correlation* assumes that many process instances share the same channel. Messages get routed to the appropriate instance by applying process-defined functions to incoming messages and then by matching the results to process instance values.
2. *Token-based Correlation* again involves many instances sharing the same channel. In this case mandatory "correlation tokens" inside the header of each message identify to which process instance they should be routed.
3. *Instance Channels* achieve correlation by using separate channels for each instance. The REST [**?**] architectural style adopts such an approach, and its claimed advantages include greater flexibility, transparency, and scalability. Zur Muehlen et. al. [**?**] describe the application of the REST architecture to BPM achieving conversations through *instance channels*. Such an approach seems intuitive to many process designers.
4. *Protocol-based Correlation* – Correlation is performed at the transport layer of the protocol. i.e. HTTP request-response, or CORBA-RPC.

WS-BPEL supports *property-based correlation* and WS-BPEL engines support *token-based correlation* through the use of implementation-specific tokens for instance routing, possibly leading to vendor lock-in.
Related to: Key Based Correlation, Property Based Correlation, Reference Based Correlation, Conversation Overlap, Hierarchical Conversation, Initiator, Follower [9], Correlation Identifier [14]
Scenario 1: Purchase Order A purchase order is received. The process sends separate queries to different suppliers for each line item. Each response is correlated over the purchase order identifier, and over the line item number. This example demonstrates the need for nested conversations.

Property-Based Message Selection Property-based message selection helps a process pick the best message off a channel. This significantly reduces the complexity logic within the process designed to iterate through an internal array of messages. There should be simpler abstractions for this.

Related to: Message Filter, Selective Consumer [14].
Scenario 2: Line Items A parts buyer wants to proceed with the best quote.

Atomic Multiple Source Consumption When messages need to be joined, from more than one source, atomic multiple source consumption greatly reduces complexity in the process model. This is because the messages from each source may need to match over a certain field or property, and packaging them together removes the need to do this in the process. Furthermore it is much simpler to design exception handling if there is only one point of failure in the process. To demonstrate this, consider the possible outcomes of consuming them separately: (1) both message arrived, or (2) one failed to arrive, or (3) both failed to arrive, or (4) they both arrived however there was a property mismatch, or (5) there were two processes and each consumed only one message. Hence, atomic multiple source consumption would any exception handling code, by allowing it to be linked to only one communication task.
Related to: Atomic Consumption [9].
Scenario 3: Purchase processing Task "*ship-goods*" will not begin until there is a confirmation of credit from the accounts department, and all line items have been notified as being "in stock".

Aggregated Consumption Consuming messages in one "pass" of a process loop greatly reduces complexity. There is no need to iterate through a set of receive actions, or to encode loop stop-conditions, which can often be a little arbitrary if the intent is to consume all available messages. Furthermore it may be necessary to choose messages only if their properties, taken collectively, satisfy certain criteria.
Related to: Aggregator [14].
Scenario 4: Shipping Company When at least 100 items have arrived destined for the same district, and a truck is available, the truck gets dispatched.

Aggregated Consumption involving Time Business processes are in many cases very time sensitive, for example the hours of business (9am to 5pm, Monday to Friday, EST). Thus there exists the need to include the notion of time into the process layer, for example by allowing message selection based on temporal constraints. Temporal constraints can typically be either absolute or relative i.e. "15 – 20 November, 2006", or "within the last 7 days". Both styles are necessary with the latter being more challenging due to the group of eligible messages being in a continual state of flux with the passage of time.
Related to: Time-Based Correlation, Moving Time Window Correlation [9]
Scenario 5: Time If, over the last five working days, more than five percent of the incoming messages that arrive at the department contain complaints then an emergency quality control process gets launched.

Contention Contention, or competition, for the same resources, is a natural phenomenon. For instance, auctions and goods tendering rely on contention between competitors. Messages, unlike auction items, have little or no intrinsic value and therefore contention over messages may not seem compelling. However, contention over messages is an enabling technique for load balancing of

message consumption; wherein many instances of the same process share the workload by only consuming messages/events when they are ready.
Related to: Competing Consumers [14].

In BPM systems supporting the deferred choice pattern [2] there are two layers of contention: (a) contention for the same message between different task instances (b) contention between tasks of the same process instance for shared control flow triggers.
Scenario 6: Competing Processes Copies of a process are distributed onto different hosts. The first process instance to claim the message provides the service, thus distributing the processing workload.

Handling Events Processes, by their very nature, need to be able to handle events that they do not solicit. Actors in the environment being under their own spheres of influence, generate events that the process may not anticipate, but the process will need to react to in a given way. Consider the following example:
Related to: Event Driven Consumer [14].
Scenario 7: Event Handling A travel booking may be canceled, should the customer decide to do so. The process needs to be able to undo certain actions at points in the process if such an event occurs, however the process never explicitly "awaits" the receipt of a cancelation event.

Channel Passing The ability for a process to "learn" about a new actor during execution by virtue of a reference to a new channel being contained in a received message. Then, the consequent ability for the process to communicate with this new actor.
Related to: Return Address [14].
Scenario 8: Channel passing A corporation begins travel booking through an agency. The agency then supplies the corporation with a channel where the status of the booking can be queried as it progresses.

Garbage Collection When messages arriving on a channel are too carefully filtered there becomes the strong possibility of having a large number of unconsumed messages on each channel. Messages have expiry dates however expiry dates cannot be updated onto a message once it is buffered on the channel. Special garbage collection filters could be added and removed from the message layer dynamically during process execution.
Related to: Message Expiration, Channel Purger [14].
Scenario 9: Garbage Collection Once a process completes all unconsumed messages addressed to this process instance are removed from the set of input channels.

Interaction Cancelation The ability to cancel incomplete interactions.
Related to: Transactional Client [14], Cancelation [2].
Scenario 10: Cancelation A supplier having posted a receive purchase order request, discovers that the warehouse needs to be replenished first, and consequently cancels its earlier receive-request.

Summary These requirements, being drawn from the respective patterns studies, ought to be supported by most state of the art solutions, however we found

that this was not the case. For instance, despite WS-BPEL being possibly the most widely accepted standard in this domain, it only provides support for the first requirement. It is one thing to support a wide range of problems in communicating business processes, however the language constructs exposed to the creators of these processes need to be suitable to their purpose, and they should be as intuitive, or conceptually aligned to their purpose as is possible.

3 Communication Model for Business Processes

In this section we introduce a model supporting interactions between processes addressing the requirements outlined above. In the proposed model, *instances* of a process model are hosted in a *process container*. The container uses a set of channels that are referred to by the process model. The channels are hosted in some form of message-oriented middleware. Channels can be used by a process container to send messages (outbound channel), receive messages (inbound channel), or perhaps both directions (bi-directional channel).

Outbound channels are composed of a unique name, possibly a type and/or an endpoint descriptor (the latter may be determined only at runtime). The proposed model abstracts away from the specific language used to describe message types. A particular embodiment of the concept of channel is one where message types are captured in XML Schema (or WSDL). Individual messages sent through the channel are then validated against its XML schema (or WSDL message type definition). Similarly, we abstract away from the mechanism used to describe destination endpoints. If using HTTP as a transport protocol for example, an endpoint can be described as a URL, while if using SOAP/HTTP, it can be described as a WSDL operation binding. Outbound channels support a range of message sending primitives which are described in detail in [5]. In the rest of the paper however, we focus on inbound channels.

Inbound channels have the same components as outbound ones but they additionally have a set of *properties*. A property is a function that takes as input a message and produces a literal value. This is similar to the concept of *property alias* defined in BPEL. However, as shown later, the scope of applicability of properties in our model is wider than that of BPEL. In BPEL, property aliases (and their composition in the form of *correlation sets*) are only used to correlate pairs of outbound and inbound messages. In contrast, in our model, properties can be used to perform other forms of message selection and aggregation.

A relation (i.e. a database table) is created for each inbound channel used in any process model. Each relation contains two predefined attributes: one of type `message identifier` and the other of type `timestamp`. Additionally, the relation contains one attribute (column) per property associated to the channel.

Properties are used to define *filters*. A filter is a function that is evaluated against the set of messages available for consumption over one or multiple channels. When the evaluation of a filter returns a non-empty set of messages, we say that the filter *matches* these messages. Filters can fulfill two purposes: (i) *garbage-collecting filters* are registered by a business process to discard unwanted

or unnecessary messages over inbound channels; (ii) *message consumption filters* are used to consume one or multiple messages from one or multiple channels. Orthogonally, filters may be *one-off* or *persistent*. A one-off filter is immediately withdrawn after it has matched a message or set of messages, while persistent filters are preserved until explicitly withdrawn.

A filter is represented as a query over the relations(s) associated with the channel(s) it refers to. These queries are always constrained to produce a relation wherein each tuple contains `message identifier` attributes.

When a message arrives onto a channel, a tuple is inserted into the relation associated with that channel. This tuple always contains `message identifier` and `timestamp` attribute values, as well as attribute values obtained by applying each of the channel's properties to the incoming message. The inserted tuple represents the incoming message for the purpose of evaluating message filters. After being abstracted as a tuple, the incoming message is either:

- Immediately routed to a message receipt action if one has registered a filter that matches the messsage (possibly in combination with other messages).
- Discarded if the message matches any of the garbage-collecting filters registered for that channel.
- Queued until it matches a garbage-collecting or message consumption filter.

Conceptually, a filter is re-evaluated every time that a new message arrives to any of the channels it refers to (or continuously in the case of filters whose query depends on the current time). In practice however, the evaluation can be made incrementally and only when required.

Primitives for registering and withdrawing filters are provided as part of the communication framework. Registration and withdrawal of filters can be initiated either by the process container or by individual process instances. When a message consumption filter is registered, the filter is run once against the set of messages available in the channels referenced by the filter. If the filter matches one or several messages, these are removed atomically from their channel(s) and given back to the process container or process instance that registered the filter. If the filter is one-off, it is withdrawn. Should no match between a filter and the existing set of messages be found upon registration of the filter, the filter is maintained and re-evaluated whenever required as explained above, until the filter is either explicitly withdrawn or it matches a message (or set of messages). Once a match is found, the message(s) are routed to the corresponding process container or instance and the filter is removed.

Garbage-collecting filters work similarly: when registered, the filter is evaluated against the contents of the channels targeted by the filter. If a match is found, the matched message(s) are discarded. If the filter is one-off, it is withdrawn otherwise, it is preserved and it is re-evaluated when required.

The proposal is formally captured as a Coloured Petri net in Fig. 2. The net shows how inbound messages are stored and matched against filters. A token in place "*incoming message*" represents a message received by the communication layer. A transition called "*put*" moves tokens from this place to a place called "*message buffer*". This latter place holds a single token containing a list of all

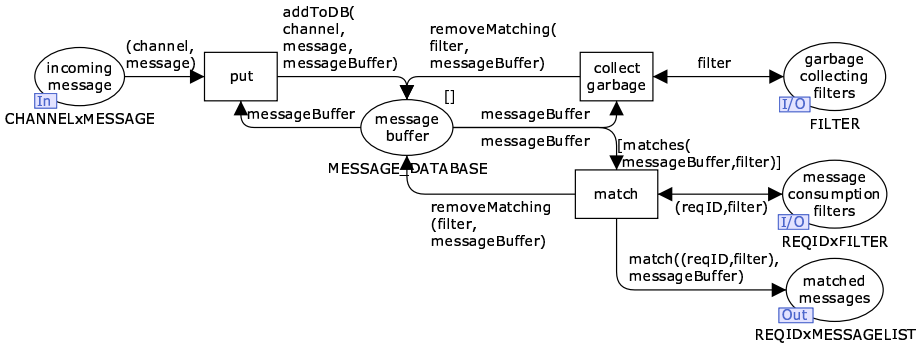

Fig. 2. Petri net capturing the treatment of inbound messages.

unmatched messages over all channels. Two of the places "*garbage collecting filters*" and "*message consumption filters*" are meant to contain one token per active filter. Transition "*collect garbage*" fires when there is a garbage-collecting filter that matches at least one of the messages in the message buffer. This transition puts back a modified message buffer in which all messages matching the garbage-collecting filter have been removed. Similarly, transition "*match*" fires if there is a consumption filter matching at least one message in the buffer. This transition also puts back a modified message buffer in which the matched messages are removed. In addition, it produces a tuple containing the filter and the set of matched messages into output place "*matched messages*". These tokens can then be routed to the process container or process instance that registered the filter in question. The latter is identified by a *request identifier* ("*reqID*"). For simplicity, the net only captures the case of "persistent filters", but it is easy to extend the net to deal with one-off filters: the only difference being that such filters should not be put back by transitions "*collect garbage*" and "*match*".

The proposed communication model abstracts away from the way channels and filters relate to business process activities or events. This way, the model can be integrated into a wide range of process definition languages. In BPEL, for example, inbound communication actions appear in two forms: as a standalone *receive* activity type and as the second leg of activities of type *invoke*, where the first leg corresponds to an outbound communication action. Thus, BPEL can be extended with the proposed communication primitives by enabling *receive* and *invoke* activities to refer to channels and filters as defined above. Channels can then be linked to *partner links* and *operations* in BPEL.

Similarly, the proposed model can be used to extend the YAWL process definition language to support a richer set of communication patterns. For example, we can define a type of message receipt task in YAWL such that: (i) upon enablement, the task registers a one-off message consumption filter defined as part of a task decomposition; (ii) the task then waits until the filter returns a match; (iii) should the task be cancelled before a match is found, the filter is withdrawn. Also, we can allow message consumption filters to be attached to the *initial condition* of a YAWL process model, to capture scenarios such as: "a new process

instance should be started whenever a given type of message (or combination of messages) has been received." An integration of the proposed communication primitives into YAWL is left as future work.

4 Implementation and Evaluation

The implementation builds on a middleware service, and API, called JCoupling (available from `www.sourceforge.net/projects/jcoupling`). JCoupling supports a superset of the communication styles supported by mainstream communication middleware. It also abstracts away from transport protocol details, thus allowing us to concentrate on the core aspects of our proposal. It supports uni- and bi-directional communication, time, space, and synchronisation decoupling, and provides support for fault propagation. JCoupling can operate over open-JMS[4] and is able to use XML/HTTP or XML/TCP for message transport.

Figure 3 presents an architectural diagram of the prototype. It shows properties and filters being used during a simple interaction between two process tasks. In Fig. 3: (1) A message is sent by task "*T1*" over channel "*Ch1*" (denoted by arrows labelled "1" going from the engine to the controller). (2) Properties "*P1*" and "*P2*" are used to extract values from the message. Next the message is put onto JCoupling (3.a), and a new tuple (row) is added to the relation (table) for channel "*Ch1*" (3.b). (4) The receiver task "*T2*" posts a filter over channel "*Ch1*". (5) Using the filter, the controller performs a query, over the relation for channel "*Ch1*". That query produces a tuple and the matching message is extracted from JCoupling (6). The callback to "*T2*" contains the message (7).

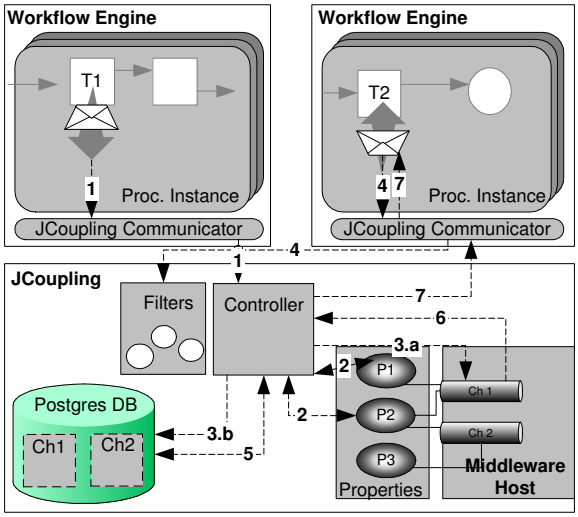

Fig. 3. Architecture of the proposal.

[4] Open JMS `www.openjms.sourceforge.net` accessed November 2006.

This section gives further details on the implementation of properties, interactions and filters, and reports on an experimental evaluation of the prototype.

4.1 Implementing Properties

The prototype implementation contains an interface called `Property`. This interface and three implementing classes are presented in Fig. 4[5]. Instances of `Property` extract various scalar values from messages. Parametric classes (i.e. "generics") are used to define the return type for the method `accessValue()`.

We envision a configuration tool for creating property instances such that process designers do not need to write Java code. For example if the process designer wishes to create an XPath property, to extract "*PurchaseOrderID's*" from messages then it would only be necessary to supply a property name, a channel binding, an XPath expression, and a Type (i.e. Text, Numeric, or Timestamp[6]) defining what type the XPath expression produces (e.g. `'PurchaseOrderId'`, `'PO_Chann'`, `'/order/@po-id/text()'`, `'Text'`).

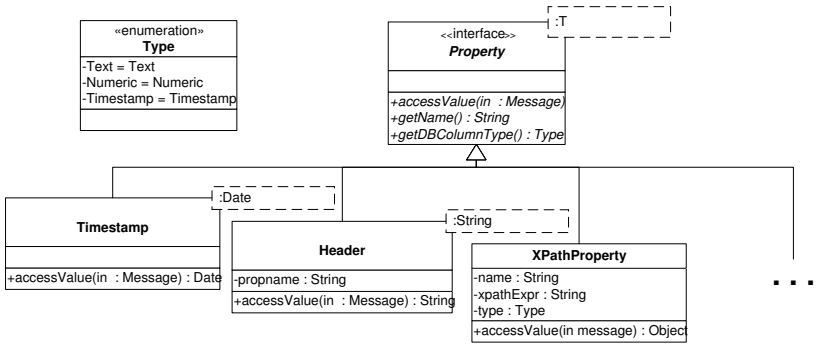

Fig. 4. UML of the property Interface.

Here is a listing of the Property interface, in Java 5.

Listing. 1. Property Interface.

```
1     ...
2     public interface Property<T> {
3     /**
4      * The name of the property.
5      * @return the name.*/
6     public String getName();
7
8     /**
9      * Retrieves the value of the property from the message.
10     * @return a type 'T' value of the property.*/
```

[5] The proposal is not limited to these three property classes. For example: an `EdiProperty` class could be created for use with EDI messages.

[6] It can be seen that in our implementation Properties can produce values of either `Text, Timestamp,` or `Numeric`. This list could possibly be extended in the future.

```
11     public T accessValue(Message message) throws Exception;
12
13     /**
14      * @return the relational attribute type for storing the property value.*/
15     public Type getDBColumnType();
16
17     /** The relational attribute types currently supported by property. */
18     public enum Type {Text, Numeric, Timestamp}
19 }
```

The prototype contains three implementations of the interface `Property`:

`TimestampProperty` inspects the message for the time it arrived on the channel. The method `accessValue(Message message)` uses the generic typing in Java to return a 'Date'. The method `getDBColumnType()` returns the enumerated type `Property.Type.Timestamp`.

`XPathProperty` extracts values out of the XML body content of the message using XPath statements passed in through its constructor. The method `accessValue(Message message)` returns an object that is either a 'Date', a 'Double', or a 'String' – depending on the `Property.Type` and the XPath statement passed into its constructor.

`HeaderProperty` extracts the values of any named properties from the message header. The method `accessValue(Message message)`, uses Java's generics to return a 'String'. The constructor receives the name of the message header property, which is used to name its corresponding relational attribute for that property, and is used to extract the value from the header.

In our implementation each channel has its own, dedicated database relation for storing its own property values that we refer to as the property relation. Adding new properties to a channel results in new columns being added to that channel's property relation. For example adding property `PurchaseOrderId` to channel `PO-Chan` causes the attribute `PurchaseOrderId` to be added to its property relation (`PO-Chan`). Any newly added attribute is declared to allow 'null' values, and hence new columns may be added during execution without causing an SQL error, if there is data in the relation. To match messages SQL queries may be executed over the property tables to find messages that match certain property values. These may be in many cases quite simple. On the other hand the SQL expressions can be quite sophisticated, as we shall see in Sect. 4.3.

As mentioned in Sect. 3, each channel's property relation has two default attributes (columns): `messageid` and `timestamp`. Table 1 presents a relation for channel `QuotesCh`.

QuotesCh			
messageid	timestamp	quote	amount
C0000M0001	2006-11-15 16:35:50	4000	1100
C1111M0002	2006-11-16 18:12:20	3000	1000
C2222M0001	2006-11-16 20:42:53	2500	1001
C3333M0003	2006-11-17 16:57:21	2000	999

Table 1. Relation corresponding to channel `QuotesCh`

4.2 Resolving Contention

A process may have to wait days before an appropriate message can be received. Hence, the prototype stores send/receive requests and performs callbacks when the interactions are complete. Indeed requests to send/receive, have their own lifecycle, including interaction cancelation (i.e. when process state change makes unnecessary an incomplete interaction). A race condition between two receive tasks is a perfect example of this. For example, a task "receive-bill-payment" and a task "receive-purchase-order-cancelation" operate such that the completion of one disables the other.

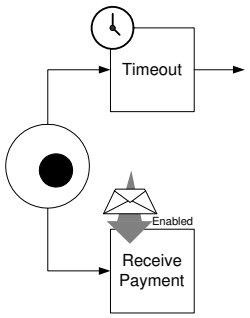

Fig. 5. Example of a YAWL process with a deferred choice involving a receive task.

Figure 5 illustrates a race condition between a timeout task and a receive task. Here, the token in the place enables two tasks, a timeout task, and a receive-payment task. Which ever task finishes first will claim the token, disabling the other, and causing the loser's unfinished action to be cancelled. The first message to arrive will make the other message unnecessary. This is a "pick" activity in BPEL, or a "deferred choice" in YAWL [1]. Accordingly the prototype (being a control layer between a process and the message layer), exposes a 'withdraw'/'cancel', primitive for send and receive interactions – also implementing the cancelation requirement (Sect. 2).

Another form of contention occurs where two tasks both want the same message/s. Contention may be a requirement (for example load balancing, Scenario 6), or it may be accidental, or perhaps even unavoidable due to the nature of the business process. Regardless contention, being a natural and sometimes necessary phenomenon, mandates a graceful approach to handling it[7].

To overcome the problems of contention within the context of our proposal we adopted the following algorithm in the prototype.

1. The receive request, containing its messages filter, is stored in the prototype.

[7] This is a distinguishing point from WS-BPEL, which throws runtime exceptions when contention between two receive-tasks occurs. Also, WS-BPEL's greedy routing of messages to process instances precludes the possibility to support contention.

2. Once the filter produces a non empty set (Ω) of message-identifiers, it is locked to ensure that the receiver (task) cannot withdraw the request.
3. Then the message engine attempts to lock every message in Ω.
4. If all messages were successfully locked, then for each message-id in Ω, that property tuple is deleted, and each corresponding message gets removed from the buffer and sent to the requestor in a callback.
5. Finally the request filter is withdrawn from the engine.

Should the receiver be unable to lock the filter at step 2 the prototype withdraws the filter. At step 3, should any of the messages in Ω be already locked then the filter locked during step 2 is unlocked, and the filter is rescheduled. Hence the first locker of the message succeeds.

When we link a business process layer to our prototype we plan to have Step 2 cause a receive task to attempt to lock its input tokens. This will prevent a receive task from locking its filter, and then receiving a message, only to find that another task has already used its enabling triggers (tokens).

4.3 Implementing Filters

This section shows how the extensions to JCoupling outlined above, support the motivating requirements of Sect. 2. Property relations, abstracting from message content, enable the use of restricted SQL queries to produce relations containing message identifiers. The primary restriction we place over the queries is that their outer projections must only be over attributes of the domain `messageid`. i.e. for all attributes of a relation produced by applying any query to the property relations, the domain of that attribute must be `messageid`.

A single property, or combinations of properties can be used to select messages. We envision that the process modeller would have a library of configurable *filter templates*, each containing semi-complete queries. The filter templates would also allow a "design time" binding to process variables, enabling runtime data to be inserted into parameterised SQL. Alternatively, for those situations requiring sophisticated aggregate operations, or complex joining expressions, the process designer may instead write their own SQL.

Property Based Selection Scenario 2 captured the need to select and proceed with the best quote, which is an example of selecting messages based on property values. The query in Listing 2 uses two properties defined over the channel "QuotesCh" (see Table 1). These are "price" and "quantity". When a receive request containing a query is invoked over the prototype, it will apply the query – returning messages in a callback to the requestor when results are found.

Listing. 2. This query combines 'price' and 'quantity' to find the best value offer.

```
1 SELECT messageid
2 FROM QuotesCh
3 WHERE quantity >= 1000
4 AND price/quantity =
5 (  SELECT min (CostPerUnit)
6    FROM (   SELECT price/amount AS CostPerUnit
7             FROM QuotesCh ) AS Q1 )
```

Conversations Scenario 1 outlined the need to correlate messages to an outer conversation for "purchase-order-ID" and an inner (nested) conversation for "line-item-id". Messages will only be correlated to a nested conversation if they satisfy correlation filters for the inner conversation, and all parent conversations. To achieve this we append an `AND-Clause` to the query. The following query (see Scenario 1) extracts runtime data from two process variables visible to the receive task: namely `PurchaseOrderID` and `LineItemID`.

Listing. 3. Achieving nested correlation through querying correlation properties.

```
1 SELECT messageid
2 FROM PoResponseCh
3 WHERE PoID = $PurchaseOrderID$
4 AND ItemID = $LineItemID$
```

We envision that the process layer will generate queries for conversations from a "conversation" construct in the process model. This construct would declare which message properties are to be used for correlation, and whether each communication task involved initialises the conversation, or follows it. Furthermore if any task involved in the conversation wants to apply message filters we may have the task append more `AND` clauses to the generated query. No proposal that we know offers this expressive power or flexibility.

We plan to introduce channel variables, and use the bi-directional channels of JCoupling to achieve Instance Channel Correlation and Protocol-based Correlation.

Atomic Multiple Source, and Aggregate Consumption Achieving a combination of atomic multiple source consumption and aggregate consumption is possible by applying a query to two or more property relations. For example Scenario 4 required aggregate consumption of 100 packages destined for the same area, and a truck availability message from another channel. Listing 4 extracts runtime data from a process level variable called `deliveryDistrict`. The query either returns at least 100 tuples, or returns nothing.

Listing. 4. This query produces a relation of messageid's wherein the attribute (`Pack.messageid`) refers to messages from Channel `Packages`, and the attribute `Truck.messageid` refers to a message on Channel `TruckWaiting`. Related messages are linked, thus removing the need to relate messages in the process.

```
1 SELECT Pack.messageid, Truck.messageid
2 FROM Packages As Pack, (SELECT messageid FROM TruckWaiting LIMIT 1) AS Truck
3 WHERE Pack.deliveryDistrict = '$deliveryDistrict$'
4 AND 100 <= ( SELECT count(*)
5              FROM Packages
6              WHERE deliveryDistrict = '$deliveryDistrict$' )
```

Aggregated Selection Involving Time Scenario 5 sought to find if at least 5% of messages that are less than 7 days old contain complaints; drawing on solutions to both time and aggregated selection. Additionally complaints were sought on *all channels*. A join between property relations will not work as every tuple from each relation represents one discrete event that needs to be considered

separately. So unlike Listing 4, the result relation will have one attribute. A `view` over the `union` of source channels (property relations), solves this. In Listing 5 this view is named "Merged". It combines properties (`messageid, timestamp`, and `complaint`) from each of the source property relations.

Listing. 5. Using `union` and `view` to combine input sources for aggregate operations.

```
1  CREATE VIEW Merged AS (
2  SELECT messageid, timestamp, complaint
3  FROM CustomerCh
4  UNION SELECT messageid, timestamp, complaint
5  FROM PartnerCh )
```

Using the above view, aggregate calculations over the messages, taken collectively, becomes feasible. Listing 6 demonstrates this. Lines 5 – 12 produce `false` unless 5%, or more, of the messages are complaints. Lines 3, 8, & 12 show the use of relative time expressions over the `timestamp` property. In cases like this we imagine that the process creator would write Listings 5 and 6 manually.

Listing. 6. This query, adapted from a continuous query in [8], produces a non-empty result of `messageid`'s when 5%, or more, of last week's messages contain complaints.

```
1  SELECT messageid
2  FROM Merged
3  WHERE timestamp  > (CURRENT_TIMESTAMP - INTERVAL '7 days')
4  AND complaint = true
5  AND (   SELECT count(*)
6          FROM Merged
7          WHERE complaint = true
8          AND timestamp  > (CURRENT_TIMESTAMP - INTERVAL '7 days')
9      ) >= (
10         SELECT 0.05 * count(*)
11         FROM Merged
12         WHERE timestamp  > (CURRENT_TIMESTAMP - INTERVAL '7 days')
13     )
```

4.4 Garbage Collection

So far we have introduced the notion of a filter that is addable to the prototype that helps in receiving messages. We extended the API slightly to have a special type of filter that removes stale messages. Like receive filters the garbage collection filters contain a set of channels, and an SQL statement designed to be applied over the property tables for each channel. The rules of execution of receive filters is exactly the as that of garbage collection filters, except that garbage collection filters do not attempt to lock or callback any receivers. They just remove messages from the channels when a match is found.

4.5 Performance Evaluation

To compare our correlation approach with that of WS-BPEL, we conducted experiments where up to ten thousand interactions were executed over our prototype and over a WS-BPEL simulator. Each experiment involved creating, in random order, at fixed intervals, a set of XML messages, all identical except for

the value of one element which was mapped to a property. Receiving processes were spawned in the same way, each of which waited for one of the created messages. The code for the experiments is released with the JCoupling distribution.

The WS-BPEL correlation simulator was built using the same middleware as our prototype. This simulator receives messages off a designated channel. Each time a message arrives, an XPath expression is evaluated against it to extract a property value (this is called a `propertyAlias` in WS-BPEL). The extracted property value is then stored in a hash table together with the corresponding message identifier.[8] Concurrently, the simulator handles requests to consume incoming messages based on property values. When the simulator finds a match between a receive request and a message, the corresponding entry is deleted from the hash table, symbolising that the message has been correlated.

Number of Interactions		**50**	**100**	**250**	**500**	**1000**	**2500**	**5000**	**10000**
Time (ms)	Proposed Approach	611	951	2230	5064	9003	25276	71409	234211
	WS-BPEL Approach	524	938	2103	4046	7825	20506	44204	133933
Performance Difference		14%	1%	6%	20%	13%	19%	38%	43%

Table 2. Results of performance tests

The test results are presented in Table 2. The table shows that our approach slightly underperforms that of WS-BPEL for small numbers of messages. This difference grows for larger numbers of messages. The accentuated difference can be explained by the fact that in the implementation of our approach, queries to match uncorrelated messages with pending receive requests are run against a persistent database, whereas in the WS-BPEL simulator, the corresponding lookup is done in-memory[9]. For larger numbers of messages, this leads to a performance penalty due to database cache management. A more consistent performance could be achieved by using an in-memory database system to implement our approach. Indeed, it is not necessary to make the correlation data structures persistent, only the messages themselves need to be persistent. In future, we plan to implement a more refined version of our approach and run a fairer and more detailed comparison against the WS-BPEL approach.

The performance penalty of our approach should be weighed against the additional functionality that it brings in. Indeed, our approach supports aggregate messaging, multi-source consumption and message contention. Moreover, as previously mentioned, there are opportunities to optimise the brute-force approach used in our implementation through incremental query evaluation.

5 Related Work

Communication in the context of distributed business processes has traditionally been researched from the perspective of *protocol* or *contract* definition. For

[8] In the interest of fairness, each entry is written to disk after being added to the in-memory hash table, as our prototype stores property values in persistent tables.

[9] Although entries are written to disk, lookups over the hash table are done in-memory.

example, the CrossFlow system [13] enables process designers to define contracts governing the communication between multiple workflows, possibly distributed across organisational boundaries. These contracts can be statically checked for consistency. Similarly, [3] proposes a method for capturing inter-workflow communication protocols and detecting deadlocks that can arise when interconnecting processes with incompatible communication protocols. This body of work is complementary to our proposal, as we do not deal with static analysis, but rather with the routing of messages to processes at runtime.

WS-BPEL exhibits strong support for conversations with the exception of *Instance Channels*, which are not supported because WSDL *ports* are not generated during process execution. WS-BPEL supports event handling and provides *some* support for channel passing. WS-BPEL, does not support the selection of messages based on their properties despite the use of properties in correlation sets. Nor does it support atomic-batched consumption. Contention over messages is not supported as messages are greedily consumed off channels and allocated to process instances immediately. Time-based message selection is not supported either. A WS-BPEL process consumes everything sent to it, thus forcing the modeller to select and throw away unwanted messages, as part of the core process logic. WS-CDL [15] has some distinguishing features, with respect to WS-BPEL, such as abstractions for channel passing and a global viewpoint over all actors in a choreography. However, in terms of the motivating requirements outlined in this paper, WS-CDL has very similar strengths and weaknesses to WS-BPEL.

Widom et al. [8, 16] propose an approach to optimising the evaluation of continuous queries over one or many data streams. They address some of the problems associated with scalability of such queries and propose incremental evaluation techniques based on the type of query. We plan to apply some of their findings to enhance and optimise the evaluation of filters used in our proposal. In particular, [16] details techniques to make continuous queries more scalable while slightly sacrificing accuracy in some cases.

6 Conclusion

This paper proposed an inter-workflow communication and control layer, to lie between traditional workflow and messaging layers – providing an isolated area for the description and execution of communication. This would provide relief from using "spaghetti" solutions to achieve non-trivial interactions. The proposal is based on a strong and relatively simple set of abstractions – namely channels, properties, property relations and filters. Channels abstract from middleware topics, and queues. Properties abstract from message content and format, while property relations provide the foundation for property filters. Filters abstract from the business level requirements for choosing and selecting messages. These enable all forms of correlation, message selection, aggregated message consumption, and time based message consumption, over a single or multiple channels evaluated collectively. The possibility of contention between process/task instances is overcome by locking messages, and filter requests. The proposal has been implemented on top of a communication API, namely JDecouple, that abstracts away from the underlying middleware and communication protocols.

Future work will aim at integrating the proposed communication abstractions into process definition languages. Specifically, we aim to extend YAWL with the ability to attach communication actions to various elements of the notation.

Further Work Considering that a message engine built using our proposal could act as a messaging hub to perhaps, thousands of process instances, there is a need to address potential issues of computational overhead. Fortunately our approach allows us to keep the datasets small, and as the number of outstanding queries increases there should be a corresponding drop in the amount of unconsumed messages, keeping the datasets small and helping the approach to scale. Nevertheless the prototype currently re-evaluates all queries when any new message arrives on any channel, or when a delta of time has elapsed (to re-evaluate any time-relative queries). The first optimisation would be to only re-evaluate a select set of the queries each time a new message arrives.

We also plan to optimise any *select-project-join* query, that does not contain a time-relative expression. The strategy relies on the fact that all existing message/s in the channel/s have already been evaluated against the query and have produced no results. Therefore if the query is over only one channel (contains no join) then it can be optimized by only applying the query over the newly arrived message. If the query is over more than one channel then we need to apply the query over the join of the tuple that represents the newly arrived message and the other relations referred to by the query. Thus greatly reducing the size of the joined result, making the query more efficient. To implement this we propose the use of database triggers that get added when a query a *select-project-join*. If the trigger produces a result then there would be a callback to the message engine, and the trigger, itself, would be withdrawn.

Acknowledgements This work is funded by ARC Discovery Grant DP0451092. The third author is funded by a Queensland Smart State Fellowship. We would also like to thank Chun Ouyang for her advice concerning event handling.

References

1. W. van der Aalst and A. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, June 2005.
2. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.
3. W. van der Aalst and M. Weske. The P2P approach to Interorganizational Workflows. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 140–156. Springer-Verlag, Berlin, Germany, 2001.
4. L. Aldred, W. van der Aalst, M. Dumas, and A. ter Hofstede. On the Notion of Coupling in Communication Middleware. In *In Proceedings of the 7th International Symposium on Distributed Objects and Applications (DOA). Agia Napa, Cyprus, November 2005*, pages 1015 – 1033. Springer Verlag, 2005.
5. L. Aldred, W. van der Aalst, M. Dumas, and A. ter Hofstede. Understanding the challenges in getting together: The semantics of decoupling in middleware.

Technical Report BPM-06-19, Business Process Management Center, Brisbane, Qld, Australia, 2006. http://www.bpmcenter.org accessed February 2007.
6. L. Aldred, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Abstractions for communication between distributed business processes. Technical Report BPM-06-28, Business Process Management Center, Brisbane, Qld, Australia, 2006. www.bpmcenter.org accessed February 2007.
7. A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. Liu, D. Knig, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language. Initial Draft of standards proposal by OASIS, June 2006. http://www.oasis-open.org/apps/org/workgroup/wsbpel/ accessed July 2006.
8. S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.
9. A. Barros, G. Decker, M. Dumas, and F. Weber. Correlation Patterns in Service-Oriented Architectures. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE), Braga, Portugal, March 2007*, pages 245–259, Springer Verlag, 2007.
10. A. Beugnard, L. Fiege, R. Filman, E. Jul, and S. Sadou. Communication Abstractions for Distributed Systems. In *ECOOP 2003 Workshop Reader*, volume LNCS 3013, pages 17 – 29. Springer-Verlag Berlin Heidelberg, 2004.
11. R. Cypher and E. Leu. The semantics of blocking and nonblocking send and receive primitives. In H. Siegel, editor, *Proceedings of 8th International parallel processing symposium (IPPS)*, pages 729–735, April 1994.
12. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
13. P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-organizational Workflow Management in Dynamic Virtual Enterprises. *International Journal of Computer Systems, Science, and Engineering*, 15(5):277–290, 2001.
14. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, MA, USA, 2003.
15. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language Version 1.0. Candidate Recommendation, http://www.w3.org/TR/ws-cdl-10/, November 2005.
16. C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 563–574, New York, NY, USA, 2003. ACM Press.