Understanding the Challenges in Getting Together: The Semantics of Decoupling in Middleware

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

[1] Faculty of IT, Queensland University of Technology, Australia
{l.aldred,m.dumas,a.terhofstede}@qut.edu.au
[2] Department of Technology Management, Eindhoven University of Technology, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl

Abstract. It is well accepted that different types of distributed architectures require different levels of coupling. For example, in client-server and three-tier architectures, application components are generally tightly coupled, both to one-another and with the underlying middleware. Meanwhile, in off-line transaction processing, grid computing and mobile application architectures, the degree of coupling between application components and with the underlying middleware needs to be minimised. Terms such as "synchronous", "asynchronous", "blocking", "non-blocking", "directed", and "non-directed" are often used to refer to the level of coupling required by an architecture or provided by a middleware. However, these terms are used with various connotations. And while various informal definitions have been provided, there is a lack of an overarching formal framework upon which software architects can rely to unambiguously communicate architectural requirements with respect to (de-)coupling. This article addresses this gap by: (i) formally defining three dimensions of decoupling; (ii) relating these dimensions to existing middleware; (iii) proposing notational elements to represent various coupling integration patterns; and (iv) proposing an API that supports all the identified coupling patterns.

Keywords: Distributed architectures, Asynchronous/synchronous operation, Distributed objects, Integration, Message Oriented Middleware, Decoupled Systems

1 Introduction

The field of distributed application integration suffers from longstanding problems. Different types of middleware have emerged over time to address these problems using various paradigms. And while existing middleware does help developers to integrate distributed applications, the diversity of paradigms adopted by different middleware has created problems on its own which can be summarised as follows: How to conceptualise and manage differences across different middleware? Indeed, a general framework for middleware remains elusive.

The heterogeneity of contemporary middleware reflects the absence of a consensus on the right set of communication abstractions to integrate distributed applications [5]. This lack of consensus can be observed even for the most elementary communication primitives, namely send and receive. As noted by Cypher & Leu: "the interactions between the different properties of the send and receive primitives can be extremely complex, and ... the precise semantics of these primitives are not well understood" [9]. One of the main issues in deriving an overarching framework for even these basic primitives is the need to support different levels of coupling.

The supported level of coupling varies from one family of middleware to another. For instance CORBA [12] is largely based on an RPC paradigm, often qualified as "tightly coupled", whereas Message-Oriented Middleware (MOM) [14] aims at supporting "highly decoupled" interactions. A correlation can be observed between the level of coupling supported by a middleware and some conceptual and technical limitations. For example, the developer of an application built on top of the Java Message Service (JMS) [13], a member of the MOM family, may find it relatively difficult to obtain synchronous responses to messages sent, even when this would simplify the coding of certain distributed transactions.

A full analysis of middleware would be a daunting task. The set of features is large, particularly when one considers, for example privacy, non-repudiation, transactions, time-outs, and reliability. This article focuses on the notion of (de-)coupling, as it is the source of many distinctions central to the design of distributed applications. Specifically, the article formulates a framework for characterising levels of coupling. The main contributions of the article are:

- A detailed analysis of the notion of decoupling in middleware and a formal semantics of various manifestations of this notion in terms of Coloured Petri Nets (CPNs) [15].
- A collection of notational elements for integration modelling. These notational elements are given a visual syntax extending that of Message Sequence Charts (MSCs) [21].
- A classification of middleware in terms of their support for various forms of (de-)coupling. This classification can be used as an instrument for middleware selection.
- A communication API supporting many coupling/decoupling integration patterns and a prototype implementation of this API.

The article is structured as follows. Section 2 establishes a nomenclature. Section 3 formalises a set of coupling dimensions while Section 4 shows how these dimensions can be composed, leading to a set of *coupling integration patterns* that provide a basis for integration modelling. Section 5 discusses how multicast and automated responses can be incorporated into the proposed dimensions. In Section 6 these patterns are used to provide a framework for tool comparison. Section 7 introduces *JDecouple*: a core middleware API inspired by the proposed framework. Section 8 discusses related work while Section 9 concludes the work.

2 Background

This section defines key terms related to coupling used in the rest of the paper.

An *endpoint* is a communicating entity – it is able to participate in interactions. It may have the sole capability of sending/receiving messages and defer processing any information to something else, or it may be able to perform both.

An *interaction* is an action through which two endpoints exchange information [20]. The most basic interaction is a uni-directional message exchange (an elementary interaction).

A *channel* is an abstraction of a message destination. Middleware solutions such as JMS, WebsphereMQ [18], and Microsoft Message Queue (MSMQ) [19] use the term "queues" to mean basically the same thing, but the crucial point here is that a channel is a logical address and not a physical one belonging to only one endpoint. Typically, more than one endpoint can receive messages off one channel. Thus the sender is able to direct a message to a symbolic destination, and the receiver may consume messages, in its own way, from the symbolic destination. Channels can be extended with many functions, including the preservation of message sequence [10, 9], authentication, and non-repudiation [14].

A *topic* is another form of symbolic destination. Like *channels* many receivers may consume messages off one *topic* - the primary difference being that all receivers get a copy of the message.

A *message* is a discrete, logical unit of information (containing a command, state, request, or an event for example) that is transported between endpoints. Depending on the technology it may contain header elements such as a message ID, timestamp, or datatype definition; or it may just contain data. It may be transactional, reliable, real-time, or delayed. It may be transported over a "channel", or even something as simple as a TCP socket.

Using the above nomenclature and following Eugster's survey of MOMs [10], three dimensions of decoupling can be identified:

- *Synchronisation Decoupling* – wherein the thread inside an endpoint does not have to wait (block) for another endpoint to be in the 'ready' state before message exchange begins.
- *Time Decoupling* – wherein the sender and receiver of a message do not need to be involved in the interaction at the same time.
- *Space Decoupling* – wherein the messages are directed to a particular symbolic address (channel) and not directly to the address of an endpoint.

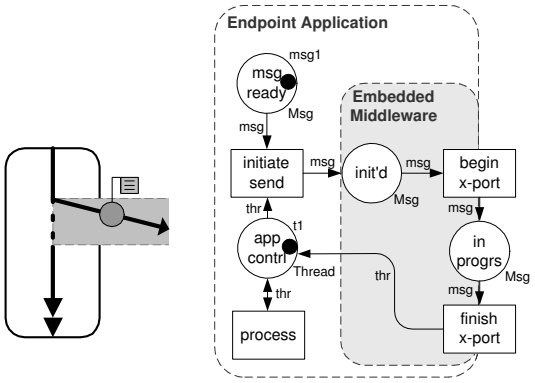
3 Decoupling Dimensions of an Interaction

In this section we provide CPNs of fundamental types of decoupling for interacting endpoints – based on Eugster's dimensions identified above. These dimensions of decoupling have relevance to a large spectrum of middleware, including MOM, space-based [11], and RPC-based middleware. CPNs were chosen for their ability to explicitly model state, parallelism, and data, their strict formal semantics and their graphical syntax. All CPNs have been fully implemented and tested using CPN Tools [7].

3.1 Synchronisation

The critical concept behind synchronisation decoupling is that of "non-blocking communication", for either, or both, the sender and receiver. Non-blocking communication allows the endpoints to interleave processing with communication from the viewpoint of the application's thread. In the following paragraphs we introduce some notational elements for various forms of synchronisation decoupling as well as a formalisation of these notational elements in terms of CPNs. While the scope of these CPNs does not include the concept of a "time-out", adding it would not overly complicate their design, although it would add some "clutter".

Blocking Send. A message send action can either be blocking or non-blocking. A *blocking send* implies that the sending application must yield its thread of control while the message is being transferred out of the local application. It does not matter if it is passing the message over a wide area network connection or to another local application. If the send blocks until the message has left the local application and its embedded middleware, it is blocking. Figure 1(b) is a CPN of a blocking send. The outer dashed line represents the endpoint while the inner dashed line represents middleware code that is embedded in the endpoint. These do not form part of the CPN language and are used only to indicate architectural concerns.



(a) Notation, blocking send. (b) CPN - blocking send.

Fig. 1. *Blocking send.* After initialising a send action, the transition "*process*" cannot fire until a thread is returned at the end of message transmission.

For readers unfamiliar with CPNs, the next two paragraphs introduce the notation using Figure 1(b) as an example. The circular nodes are called 'places' (e.g. *"msg ready", "app contrl"*, and *"in progrs"*), and they represent potential states. The rectangular nodes are called 'transitions' (e.g. *"initiate send", "begin x-port"*, and *"process"*), and have the potential to change any CPN from one state to another. Tokens, e.g. the black dots in the places *"msg ready"* and *"app contrl"*, represent a particular state of the model. Figure 1(b) is in a state where

a message is ready to be sent, and the application has control of its own thread. This is the initial state of the CPN, and is referred to as its 'initial marking'. According to the firing rules of CPNs two transitions are concurrently 'enabled' (*"initiate send"* and *"process"*). When either of these transitions fire it will consume one token from each of its 'input places' and produce one token into each of its output places. Transitions are only enabled if *all of their input places contain one token*. Therefore, if the transition *"initiate send"* fires the transition *"process"* cannot fire until a token returns to its input place (*"app contrl"*).

In Figure 1(b) the places are type-constrained. For instance the place *"msg ready"* can only hold tokens of type `Msg` – shown as an annotation to its bottom-right side. The annotations shown at the top-right side of the places *"msg ready"* and *"app contrl"* are references to constant values. The values are used to determine the initial marking of Figure 1(b). For instance the annotation *"msg1"* is a constant value of type `Msg`. Its presence puts a token into the place *"msg ready"*. Each arc entering/leaving a place in this CPN is annotated by a variable – typed according to the place the arc connects to. Arcs leaving transitions can optionally invoke ML [17] functions. This feature will be used in Section 3.3.

In Figure 1(b), when a message is ready (represented by a token inside the place "*msg-ready*") and the application is ready (represented by a token inside the place "*app-contrl*") the endpoint gives the message to the embedded middleware.[1] The endpoint yields its thread of control to the embedded middleware, getting control back once the message has completely left the embedded middleware. Inside the embedded middleware the transitions "*begin-x-port*", "*fin-x-port*", and the place "*in-progress*" are placed over the edge of the endpoint. This denotes that the remote system (receiver endpoint or middleware service) will bind to the sender by sharing these transitions and the place. The assumption is that inside the middleware, at a deeper layer of abstraction, systems communicate in a time-coupled, synchronisation-coupled manner, regardless of the behaviour exposed to the endpoint applications. Therefore this CPN may be "transition bounded" with remote systems.[2]

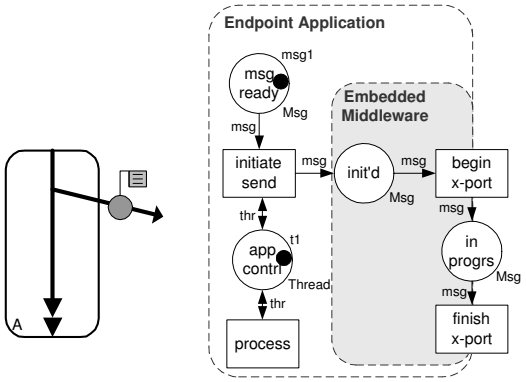
In a blocking send there is a synchronisation coupling of the sender application (endpoint) with something else – but not necessarily the receiver as we will show in Section 3.2.

Non-blocking Send. *Synchronisation decoupling* is observable at the sender in the form of a *non-blocking send* [10]. A non-blocking send means that message transmission and local computation can be interleaved [23]. Figure 2(a) presents a notation for non-blocking send, based on the MSC notation [21]. Figure 2(b) defines the concept in CPN form. This Figure, like that of blocking send (Figure 1) is transition bounded with remote components through the transitions in

[1] In this series of CPNs we represent tokens as a black dot. This is not strictly necessary as the initial markings are shown textually. It is however, a convention we adopt that is intended to assist their readability.

[2] "Transition bounded", in this context, means that two distributed components share a transition (action), and must perform it at exactly the same moment.

the embedded middleware of the application. Snir and Otto provide a detailed description of non-blocking send [23].



(a) Notation, non-blocking send. (b) CPN of non-blocking send.

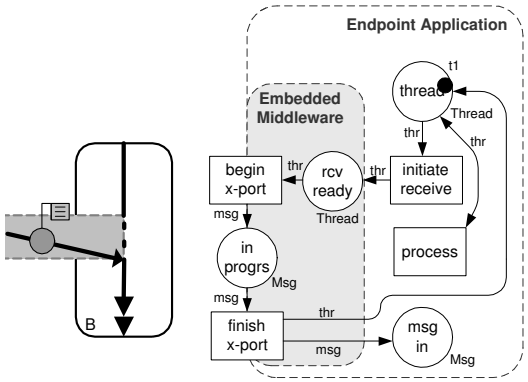
Fig. 2. *Non-blocking send.* The transition "*process*" can be interleaved with communication because a thread is not yielded to the embedded middleware.

A non-blocking send is a necessary condition, but not a sufficient condition to achieve total synchronisation decoupling, which is to say that the receive action must also be non-blocking. If both send and receive are blocking (non-blocking) then a total synchronisation coupling (decoupling) occurs. A partial synchronisation decoupling occurs when the send is blocking and the receive non-blocking, or vice-versa.

The non-blocking send is a fairly uncommon feature of middleware solutions. For instance all RPC-based implementations use blocking send. MOM implementations such as "Websphere MQ" [18] and MSMQ expose only a blocking send operation for passing messages onto the middleware service. This middleware service can optionally be deployed onto the local host, meaning that the send operation only blocks while the message is passed between applications on the same machine. Hence the sender does not block while the message travels through the network. However, this is not always possible, or practical due to licensing limitations, or the limited computational power of a small device. We propose that such middleware could be improved by exposing an explicit non-blocking send in the API.

Blocking Receive. Like message send, message receipt can either be blocking or non-blocking [9]. The definition of *blocking receive* is that the application must put a thread into a waiting state in order to receive the message (ownership of the thread is usually returned to the endpoint when the message is received). This means that the receiving application is synchronisation coupled to either the message sender or the middleware service (depending on whether the blocking action waits for an event on the middleware, or the sender). Section 3.2

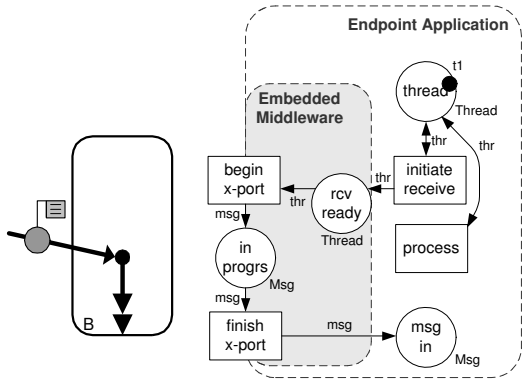
formalises this notion. Figure 3 presents a notation and model for blocking receive. The transition *"initiate receive"* of Figure 3(b) is fired the instant the receiver 'intends' to begin waiting for the message, this may even occur before the message is sent.



(a) Notation, blocking receive. (b) CPN of blocking receive.

Fig. 3. *Blocking receive.* A thread must be yielded to the embedded middleware until the message has arrived.

Non-blocking Receive. The *non-blocking receive* occurs when the application can receive a message, without forcing the current thread to wait. This is illustrated in Figure 4.



(a) Notation, non-blocking receive. (b) CPN of non-blocking receive.

Fig. 4. *Non-blocking receive.* A thread need not be yielded to the middleware in order to receive.

A well-known embodiment of non-blocking-receive is the event-based handler described in JMS [13]. Once a handler is registered with the middleware it is called-back when a message arrives. The Message Passing Interface (MPI) pro-

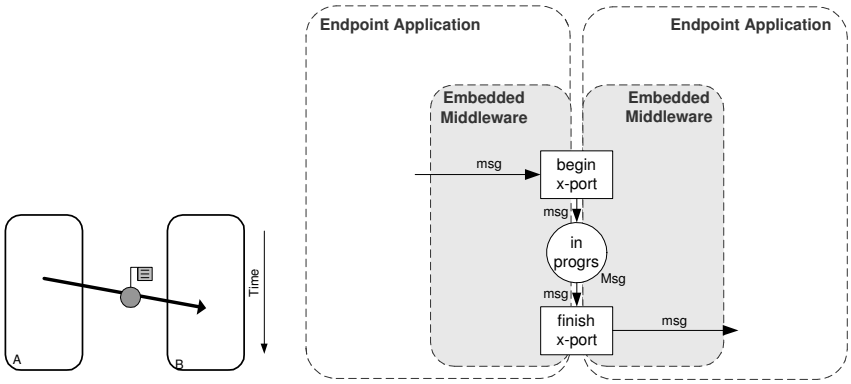
vides another embodiment of non-blocking receive that is not event-based [23], wherein the receiver polls for the presence of a new message.

Non-blocking receive, as an integration abstraction, seems less popular than blocking receive. This is probably because blocking receive interactions are simpler to program and debug [14]. One frequently observes statements about the merits of an asynchronous architecture. While such statements may be valid, they usually refer to the value of time-decoupled systems, not synchronisation-decoupled systems. Therefore the synchronisation dimension presented in this section is orthogonal to the general usage of the word "asynchronous" in the integration community.

3.2 Time

The dimension of time decoupling is crucial to understanding the difference between many peer-to-peer middleware paradigms and server-oriented paradigms (e.g. MPI versus MOM). In any elementary interaction time is either coupled or decoupled.

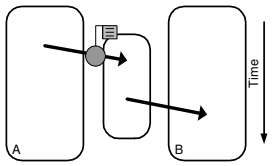
Time-Coupled. *Time-coupled* interactions are observable when communication can not take place unless both endpoints are concurrently connected. In time-coupled systems the message begins by being wholly contained at the sender endpoint. The transition boundedness of endpoints can guarantee that the moment the sender begins sending the message, the receiver begins receiving. The concept is presented in Figure 5 wherein the endpoint applications are joined directly at the bounding transitions ("*begin x-port*" and "*fin x-port*"). Time-coupled interactions accord with the general usage of the word "synchronous". Figure 5 does not show the end-points. The "sends" and "receives" may be blocking or non-blocking. Hence, Figure 5 should be seen in conjunction with the earlier figures.



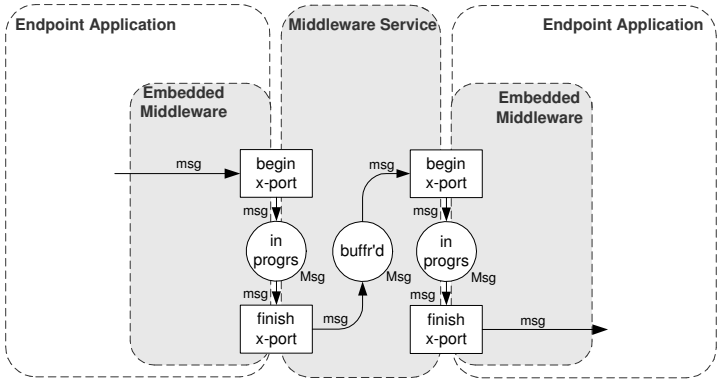
(a) Notation, time coupled. (b) CPN of time coupled messaging.

Fig. 5. *Time coupling* is characterised by transition-bounded systems.

Time-Decoupled. *Time-decoupled* interactions allow messages to be exchanged irrespective of whether or not each endpoint is concurrently operational. To achieve time decoupling a third endpoint is needed, that both the sender and receiver can access. Time-decoupling is presented in Figure 6. The two endpoints, and the middleware service are again transition bounded. However now, the middleware service is able to buffer the message, as captured by the place "*buffr'd*".



(a) Notation, time decoupled.



(b) CPN of time decoupled messaging.

Fig. 6. *Time decoupling* is characterised by the presence of an intermediate endpoint.

MOM solutions such as Websphere MQ and MSMQ can be deployed in a "hub and spoke" arrangement, which is truly time-decoupled. An alternative arrangement is the "peer to peer" topology. In such a topology the sender endpoint and middleware service are deployed on the same host. This latter topology may seem time-decoupled, but it is not because interactions can only take place if both hosts are concurrently connected to the network. This may be a problem for endpoints on hosts with unreliable network connections, e.g. mobile devices.

3.3 Space

Space is the final dimension of decoupling considered in this taxonomy.

Space Coupled. For an interaction to be *space coupled* the sender uses a direct address to send the message to. The sender has information that identifies the

receiver endpoint uniquely within its environment. This can be, for example, location data, or other data that can be resolved into location data. Figure 7 presents the concept of space coupling. Our CPNs of the space dimension are incorporated together, and presented at the end of this section. Space-coupled interactions always involve one sender interacting with *one of one* receiver.

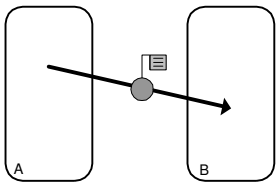
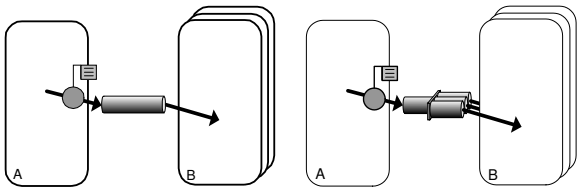


Fig. 7. *Notation, space coupled.* The sender directly addresses the receiver.

Space Decoupled. *Space decoupled* interactions on the other hand allow a sender to send a message without requiring explicit knowledge of the receiver's address. Decoupling in space generally makes architectures more flexible, and extensible.



(a) Channel - the first receiver gets the message.

(b) Topic - all receivers get the message.

Fig. 8. Extensions to MSCs representing two forms of space decoupling.

There are two distinct forms of space-decoupling. Space-decoupled architectures permit one sender to interact with *one of many* receivers (over a channel) or *many of many* receivers (over a topic), without uniquely identifying the receiver. Figure 8 introduces extensions to MSCs that present notations for "channel" and "topic" based interactions.

In Figure 9, the transition *"begin x-port"* decomposes to the sub-net in Figure 10.[3] This captures the three options of the space dimension into the CPNs. The places *"new subscr"*, *"subscriptions"* and transition *"subscribe"* model the ability for receiver endpoints to subscribe to destinations. Each receiver has its own unique application ID (type `Apid`). Hence a new subscription token (i.e. *"apID, dest"*) is an endpoint/application ID combined with the relevant destination. Transition *"subscribe"* takes this token and appends it to the list of subscriptions represented by the token in place *"subscriptions"*.

[3] Sub-nets can be used to hide details. Specifically, a transition can be decomposed into a sub-net and by replacing this "substitution transition" by its decomposition one obtains its semantics.

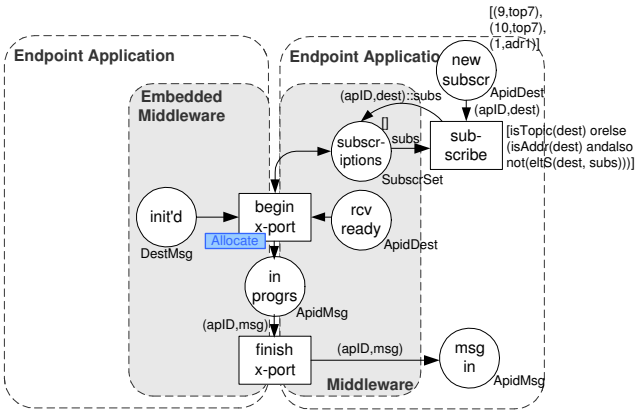


Fig. 9. CPN presenting a semantics for the space dimension. The transition *"begin x-port"* has been annotated with a box labelled *"Allocate"*, which means that this transition decomposes to a sub-net (see Figure 10).

In the CPN of Figure 9 any topic, channel, or address can be bound to a message and put into the place *"init'd"* (which is typed `DestMsg`). In order for any message receipt to occur, a token must enter the receiver's place *"rcv ready"*. This place is typed `ApidDest` and it identifies the receiver's application ID and the destination to consume the message from.

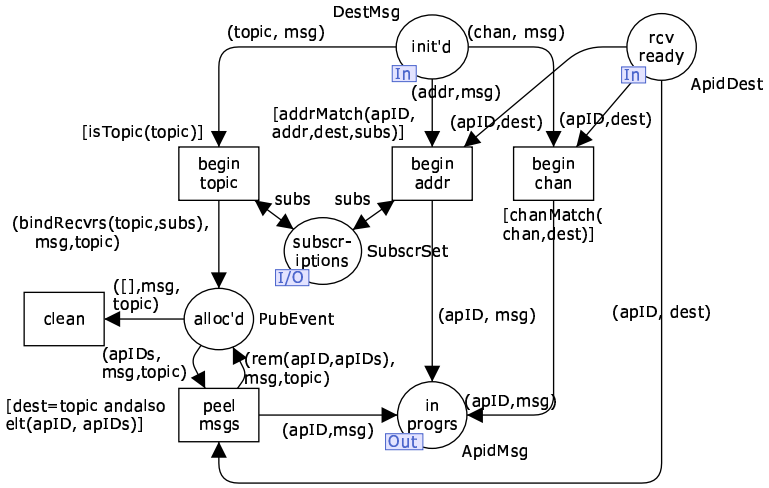


Fig. 10. This CPN is the sub-net of transition *"begin x-port"* in Figure 9.

Figure 10 presents the sub-net decomposition of the transition *"begin x-port"* (from Figure 9). This CPN distributes each message to the right receiver/s, in accordance with the expected semantics of the destination type (i.e. topic, address, or channel). The input places (*"init'd"* and *"rcv ready"*), the output place (*"in progrs"*), and the input/output place (*"subscriptions"*) correspond to the similarly labelled places of the parent net (Figure 9). Channel bound messages will only fire the transition *"begin chan"* when there is a token in each

input place that identify the same channel. The transition *"begin addr"* fires when the input tokens refer to the same address, and the application id (*"aID"*) exists in the subscription set (*"subs"*)[4]. Firing the transition *"begin topic"* only requires an input token identifying a topic-message, then the application IDs in the subscription set *"subs"* (a set of <application-ID, destination> pairs) that are subscribed to that that topic effectively are allocated a copy of the input message. Each subscriber can then obtain, or peel off, its copy of the message using the same technique as for addresses and channels, i.e. by putting a token into the input place *"rcv ready"*. Hence while the interface for each of the three options of space coupling are consistent, the way messages get bound to application-IDs is different.

Thus, direct addressing supports interactions between a sender and *only one* receiver. Channels send the message to *one of many* receivers, and topics to *all of many* receivers.

Having used a formal modelling technique to demonstrate these different behaviours with the same interface, of course has several uses: Firstly, it improves the conceptual rigour of implementations based on these models (see Section 7), secondly it can provide the opportunity to mathematically prove some properties about integration.

3.4 Summary

The dimensions of decoupling included "synchronisation decoupling" (with its four options), "time decoupling", and "space decoupling" (with its three options). Each dimension has its own precise behaviour and semantics, and were introduced in a graphical notation based on MSCs, and more precisely as CPNs. The CPNs shared similar structure and place names, which provides a hint of how to combine these dimensions together, while preserving their individual semantics.

4 Combining Synchronisation, Time, and Space

We contend that the dimensions of decoupling presented in the previous section are orthogonal to each other. Hence models integrating all aspects may be composed from options along each of the three dimensions, and so can the CPNs of Section 3 that capture the semantics of the individual dimensions. These combined models then define a precise overall behaviour.

The set of achievable combinations, of these dimensions, can be used as a palette of ways to couple/decouple systems and can thus be applied to an integration problem or to the selection of an appropriate middleware product.

[4] We assume that the process of an application claiming an address is related to a receiver subscribing to a topic; the difference being that only one application can ever claim an address for any state of the CPN. Hence we consider an address not to be a hardware address necessarily but a datum identifying the receiver.

Any option (of the four) for synchronisation can be combined with any option (of the two) for time, which in turn can be combined with any option (of the three) for space. Hence, for one-directional messaging, there are twenty four ($2^2 * 2 * 3 = 24$) possible ways to combine these three dimensions, each with its own precise behaviour.

Composing Petri nets. Our attempts at combining these dimensions, using CPN Tools, have indicated the dimensions are indeed orthogonal. Therefore any CPN from each of the dimensions can be combined with any CPN from any other dimension. For all possible (de-) coupling combinations, the semantics and behaviour are identical to those of each dimension's CPN in isolation. Furthermore the behaviour of the combined CPN still exhibits the behaviour implied by its constituent source CPNs. This means one can create integration models that are precise, and unambiguous thereby allowing a degree of strong separation between a model of integration, and the technologies used to achieve it.

Merging these CPN requires minor modifications. Due to space constraints we will present the modifications we made to create two of the twenty four possible merged CPNs. The other twenty two possible CPNs can be created similarly.

Figure 11 models a synchronisation-coupled, time-coupled, space-coupled combination. Our first step was to start with the blocking send (Figure 1(b)) and blocking receive (Figure 3(b)), and join these two CPNs together along their similarly labelled nodes *"begin x-port", "in progrs"* and *"finish x-port"*. This merged CPN is time-coupled, and therefore we do not add any intermediate detail between the blocking send and the blocking receive.

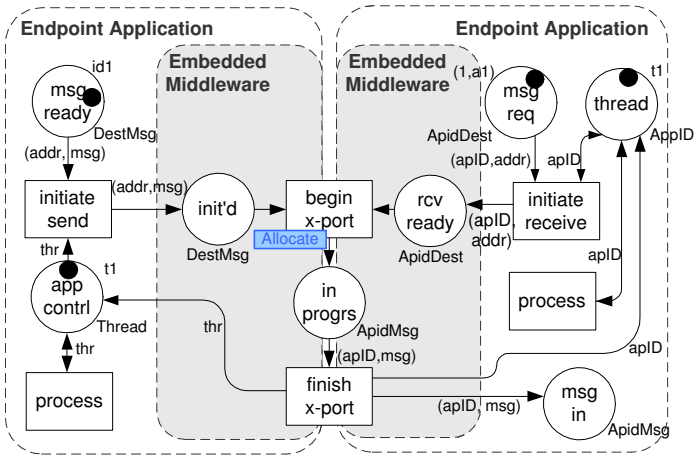


Fig. 11. Petri net of a synchronisation-coupled, time-coupled, space-coupled interaction between endpoints.

To produce Figure 11 we changed the types of selected places, and arcs to account for the space dimension. We extended the type (from `Msg` to `DestMsg`) for the places *"msg ready"* and *"init'd"*, and likewise extended the arc variables. We also extended the type (from `Thread` to `ApidDest`) for the place *"rcv ready"* and

a new place *"msg req"*. This represents a type identifying the receiver (`Apid`) and the Channel/Topic/Address (`Dest`). The place labelled *"thread"* was extended from `Thread` to `Appid`, and the places *"msg in"* and *"in progrs"* were extended from `Msg` to `ApidMsg`. The final step we performed to create Figure 11 was to add the sub-net *"Allocate"* (Figure 10) to the transition labelled *"begin x-port"*.

A second example of merged CPNs is presented in Figure 12. It models a synchronisation-decoupled, time-decoupled, space-decoupled (channel) interaction pattern. Being time-decoupled we added a "middleware service" between the sender and receiver, and stitched the boundary nodes (transitions and places) of the sender and receiver to this service.

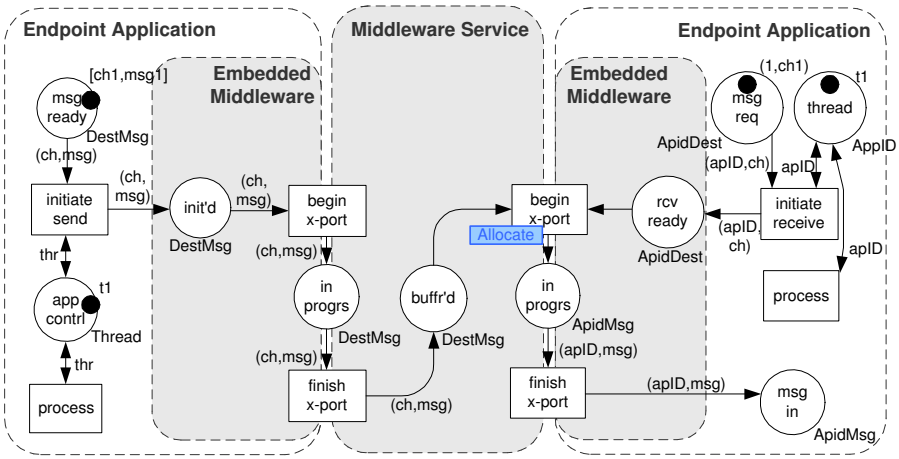


Fig. 12. CPN of a synchronisation-decoupled, time-decoupled, and space-decoupled interaction.

In Figure 12 we extended the type of selected places of the sender and receiver using the same approach presented for creating Figure 11. The type for the places *"buffr'd"* and the sender side *"in progrs"* was also extended (from `Msg` to `DestMsg`). Finally we added the sub-net *"Allocate"* to the transition labelled *"begin x-port"*. However, being time decoupled, the modification is only performed to the transition on the receiver side.

In summary we have presented a method for binding a sender CPN to a receiver CPN, and an optional, intermediate buffer CPN (used if the systems are time-decoupled).

Graphical Notation of Compositions. To create a graphical notation for the twenty four combinations we merge the graphical notations from each option of a dimension (from Section 3) with each option from each of the other two dimensions. Figures 13, 14, and 15 present these. These graphical illustrations are further extensions to MSCs [21], and are obtained by "overlaying" the

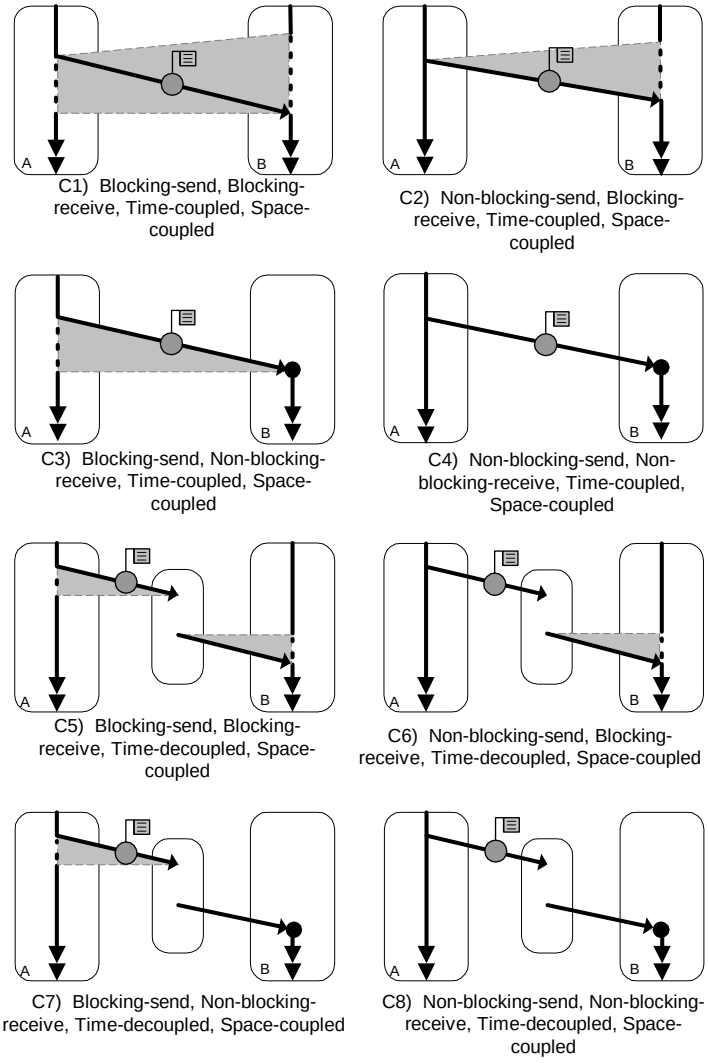


Fig. 13. Notations for the coupling integration patterns – space-coupled communication.

notations for decoupling introduced in Section 3. They are numbered from one to twenty four, and represent different patterns of coupling integration.

We contend that this set of coupling integration patterns can be used in defining requirements during system analysis and design. Each pattern has its own specific, unambiguous behaviour. Furthermore they are sufficiently different that some will be more suitable to a given integration problem than others. For instance a multiplayer real-time strategy game would not have much use for time-decoupled interactions.

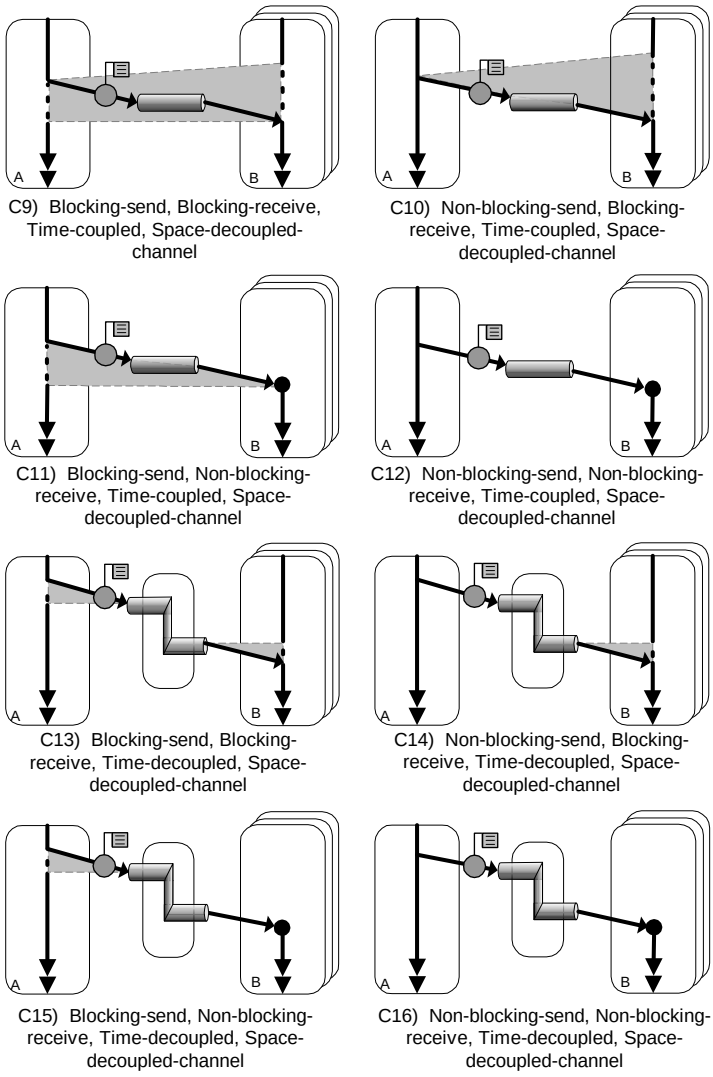


Fig. 14. Notations for the coupling integration patterns – space-decoupled channel.

Example. Consider a hospital that needs to integrate a Business Process Management (BPM) system and a proximity sensor system to send requests to medical staff based on their skills and location. Each medical staff is given a mobile device that relays location information to a central system. The BPM system uses this information to allocate work items to perform patient care services in an efficient, timely manner.

The challenge is to design a conceptually clean, integration model accounting for the varying levels/types of connectivity between distributed systems, and mobile resources. Clearly the mobile devices will not always be connected to the central system (due to varying levels of signal availability), and therefore the use

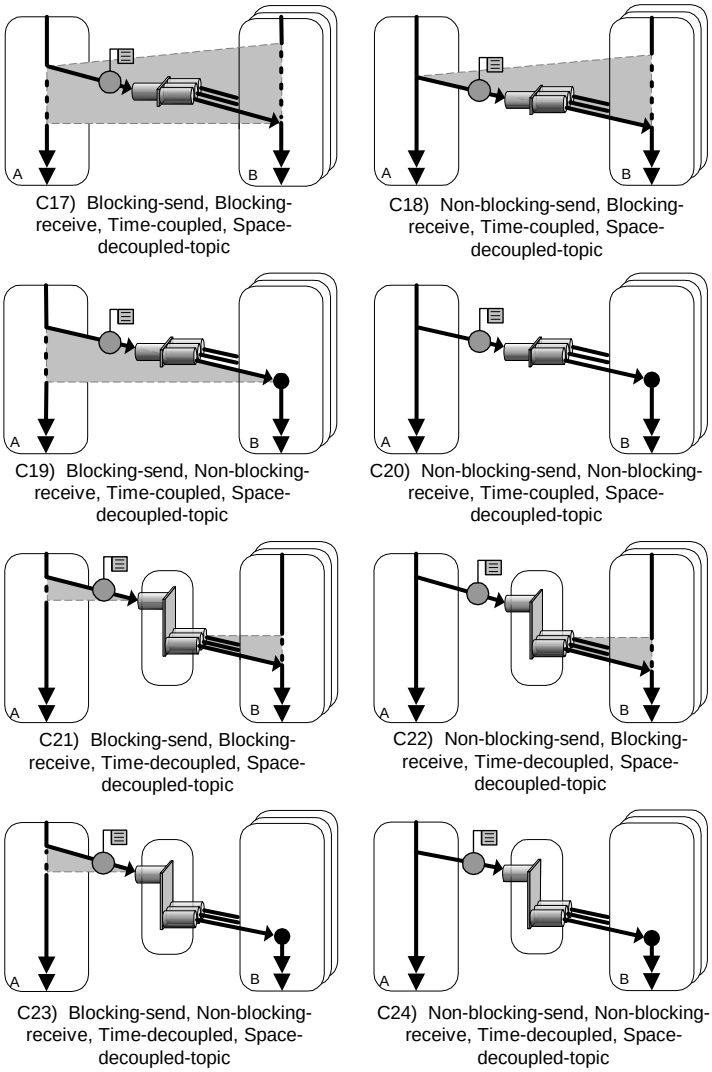


Fig. 15. Notations for the coupling integration patterns – space-decoupled topic.

of non-blocking send is advisable. Hence messages to and from mobile devices could be stored until the signal is restored. New mobile devices might need to be added to the system and device swapping may occur – which should not break the integration. Therefore space decoupling is required, but we do not want to notify many instances of the same resource with the same work request, thus ruling out publish-subscribe. Finally, it is likely that the mobile devices have intermittent connectivity and therefore time decoupling between mobile devices and the central system is necessary.

Based on these requirements it is clear that one should use coupling integration pattern '14' (Non-blocking-send, Blocking-receive, Time-decoupled, Space-

decoupled-channel) or pattern '16' (Non-blocking-send, Non-blocking-receive, Time-decoupled, Space-decoupled-channel) for the hospital integration project.

5 Beyond the Dimensions

Some aspects of integration, such as multicast, and different forms of response constitute equally fundamental requirements for integration problems. Hence their relationship with the dimensions of coupling are described in this section.

5.1 Multicast and Message Joining

Multicast involves sending a message to many receivers. This is orthogonal to the space dimension, and in particular, publish subscribe. Indeed multicast corresponds to a set of interactions to several destinations. It can be achieved by performing several interactions in parallel or any order. Each of these interactions may have as a destination, an address, an channel, or a topic.

The converse to multicast is the multiple message join. Essentially messages from many senders get served to the receiving application/endpoint as an aggregated batch. Once again, this is orthogonal to the dimensions of (de-) coupling. Support for multicast and message join are useful features of a messaging solution. Thus they are strongly related but do not cut into the coupling dimensions.

5.2 Supporting response and fault notification

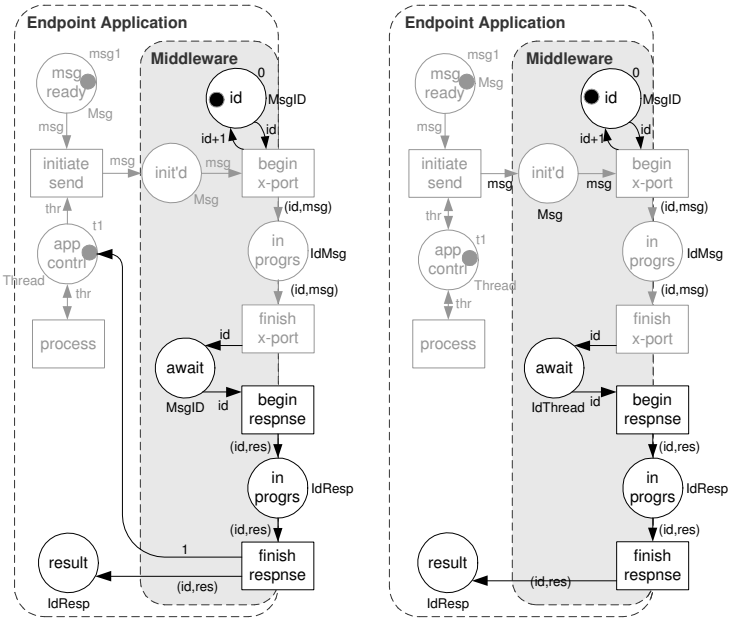
The analysis of (de-) coupling has thus far only accounted for uni-directional messaging. Since many middleware solutions support both uni-directional and bi-directional messaging we consider the investigation of coupling, in the context of bi-directional messaging, to be essential.

Bi-directional messaging, and specifically request-response interactions, encompass three additional concepts on top of those presented before: delivery receipt, response, and application-level fault. A delivery receipt (i.e. system acknowledgement) is a message emitted by the middleware to indicate to the sending endpoint, that its message has been successfully received. A delivery receipt does not imply that the targeted endpoint has processed the message – just that it has received it. A response is a message from endpoint E2 to an endpoint E1, following the receipt and processing of a previous message from E1 to E2. Finally an application-level fault is a special type of response indicating that an error occurred during the processing of the first message by E1. The CPNs presented here do not deal with networking faults, only with application level faults.

RPC-based middleware supports a "solicit-response" message pattern for the client, and a "request-response" message pattern for the server. However, highly decoupled systems support request-response interactions and fault notification to varying degrees. It can be observed that middleware solutions and standards generally amalgamate request-response patterns with time-coupled, synchronisation-coupled patterns of interaction. For example Axis [3] (being time-coupled, and using synchronised sender) supports responses, as is

CORBA [12]. On the other hand time-decoupled solutions such as Linda [11], and MSMQ provide no direct support for request-response interactions.

There are some minor exceptions to this general observation, such as the JMS API, which through its QueueRequestor and TopicRequestor, provide some support for request-response interactions over a queue (channel) and a topic respectively. However the responder must explicitly extract the return queue/topic from the request, and use it to direct the response. These classes do not support propagating receiver exceptions and the TopicRequestor returns only the first response and ignores the responses of all other subscribers.



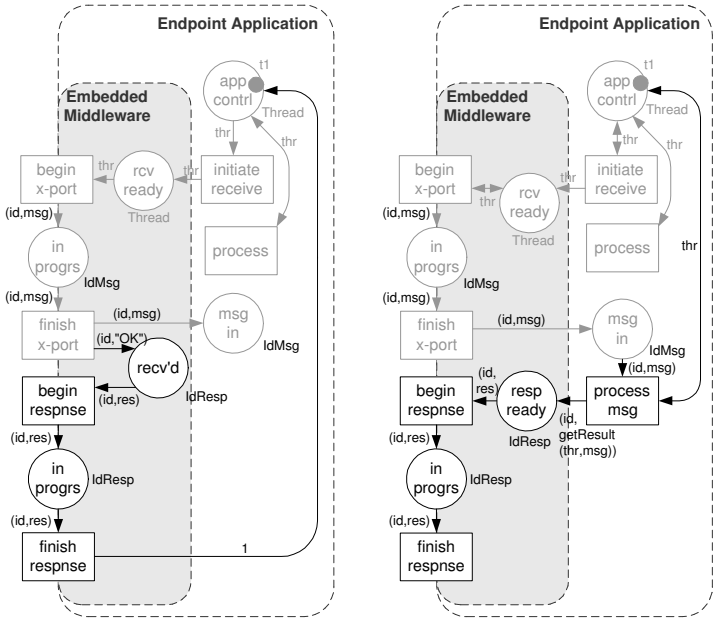
(a) CPN of blocking send with response, or delivery receipt.

(b) CPN of non-blocking send with response, or delivery receipt.

Fig. 16. Extended CPNs dealing with synchronisation and sending. The changes made to the related CPNs from Section 3 are black, while the unchanged elements are grey.

Therefore users of time-decoupled solutions are typically forced to use workarounds to implement request-response interactions. The designers of time-decoupled solutions appear to overlook these types of interactions despite the fact that they are an essential requirement for many forms of distributed computing. The CPN models from Section 3 covering synchronisation and time were extended to *optionally* support request-response interactions, fault notification, and/or delivery receipt. These extended CPNs (see Figures 16 and 17) indeed preserve the original orthogonality of time, space, and synchronisation. In Figure 16 the structure of the boundary nodes (*"begin x-port"*, *"in progrs"*, *"finish x-port"*, *"begin respnse"*, and *"finish respnse"* in either CPN is identical. The major difference being that Figure 16(a) waits for the result, whereas Fig-

ure 16(b) continues processing immediately. The application in Figure 16(b) can rendezvous with the result when it is ready.



(a) CPN of blocking receive with delivery receipt.

(b) CPN of non-blocking-receive with response.

Fig. 17. Extended CPNs that deal with synchronisation and receiving in a request-response interaction. The changes made to the related CPNs from Section 3 are black, while the unchanged elements are grey.

One can observe that the alternative CPNs of Figure 17 do preserve their blocking and non-blocking behaviour respectively. The CPN for blocking receive (Figure 17(a)) includes the return of a delivery receipt, whereas non-blocking-receive (Figure 17(b)), includes the return of a response/fault. A delivery receipt is not intrinsic to blocking receive, just as responses and fault notification are not intrinsic to non-blocking-receive. They are presented in these CPNs as alternatives, a choice more inspired by expediency.

We have seen that it is necessary to extend the CPNs for synchronisation in order to cover responses. It is also necessary to extend the CPNs for the dimension of time. Figure 18 (request, optional response, time-decoupled) shows that the response is generated by the intermediate point of the interaction. This means that for time-decoupled interactions the semantics that two systems can interact without being active concurrently is preserved. The place *"resp ready"* stores and returns a signal to the sender indicating the message has been buffered, and is ready for the receiver to retrieve. If the case arises that the sender still wishes to retrieve a response from the actual receiver in a time-decoupled manner, the CPN of Figure 18 allows for this by providing an optional response polling service. This is started at the place *"polling"* and continues through transition *"bgn td*

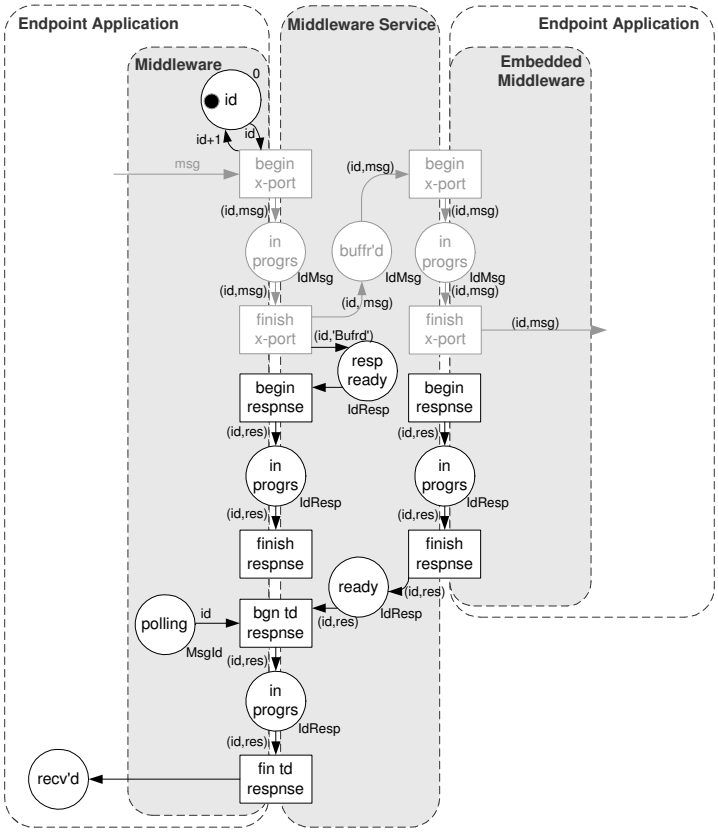


Fig. 18. Time decoupling, extended to cover request-response interactions.

respnse". We do not include the extended CPN for "time-coupled" interactions, because it is a trivial extension of Figure 5(b).

We consider that the use of the response mechanism presented here should be optional. Implementations that provide this range of integration services should not force users to use, or even retrieve responses. However it would seem sensible if implementations, like the CPNs consistently performed responses, and simply left it optional for the clients to retrieve them.

We have concluded that in fact delivery receipts (system acknowledgements), responses, and faults can be added to the semantics of blocking/non-blocking, time-coupled/time-decoupled topologies without interfering with their original semantics, as defined in Section 3. Therefore it would be useful, when comparing middleware solutions in terms of their coupling, to also take into account their relative support for alternative patterns of response. Based on our models and our survey of middleware solutions/standards, these include:

– Preprocess acknowledgement – a signal, provided by the middleware, is returned to the sender indicating the successful receipt of the message.
– Postprocess acknowledgement – a signal, provided by the middleware, gets returned to the sender indicating that the message was successfully processed.
– Postprocess response – a response, containing information provided by the receiver, gets returned to the sender.
– Postprocess fault – an exception/fault occurs in the receiver while processing the message, and information about this gets propagated back to the sender.
– Blocking-receive acknowledgement – the sender blocks for the response.
– Non-blocking receive acknowledgement – the sender thread can use a non blocking technique to receive the acknowledgement/response message.

The options for responding to a message do not interfere with the semantics of coupling and decoupling. For example, time-decoupling enables, but does not force "fire and forget" interactions. Likewise, a blocking-send does not imply a response, and a non-blocking send does not imply the lack of one. The primitives of coupling and their formal semantics help clarify this orthogonality despite the support (or lack thereof) by solutions and standards.

6 Comparison of Middleware Solutions and Standards

The twenty-four coupling integration patterns identified in Section 4 can be used as an instrument for evaluating middleware solutions and standards – in terms of their support for different forms of (de-) coupling. To illustrate this proposition, we have evaluated the following: Java-spaces[5], Axis [3], CORBA [12], JMS[6], Websphere MQ[7], MSMQ [19], MPI[8], and the JDecouple API (introduced later in this article).

In most cases the documentation (as opposed to products) for these standards and solutions was used as a guide to determine their degree of support for these patterns. The results of this evaluation are presented in Table 1. Solutions that directly support a pattern are given a plus ('+'). Those able to support a pattern using minor work-arounds are given a plus minus ('+/–'), and those requiring greater effort are assigned a minus ('–'). A detailed rationale behind these assessments is provided in Appendix A.

The coupling integration patterns (C1 - C24) correspond to the patterns in Figures 13, 14, and 15. In addition there are two patterns related to multicast (see Section 5.1) and six patterns related to the generation and handling of responses (see Section 5.2).

The hospital scenario from Section 4 requires either patterns C14 or C16. The table shows that only JMS, Websphere MQ, and MSMQ provide any support for patterns C14 and C16, and their support is only partial.

[5] Java Spaces `http://java.sun.com/developer/products/jini/index.jsp`, accessed June 2006.
[6] J2EE-SDK V. 1.4, `http://java.sun.com/products/jms/`, accessed June 2006.
[7] Websphere MQ V 5.1, [18].
[8] MPI Core: V. 2, [23].

		Java Spaces	Axis	CORBA	JMS	Websphere MQ	MSMQ	MPI	JDecouple
Coupling Integration Patterns	Ptrn ID								
BlkSndBlkRec TmeCpl SpcCpl	C1	–	–	–	–	–	–	+	+
NBlkSndBlkRec TmeCpl SpcCpl	C2	–	–	–	–	–	–	+	+
BlkSndNBlkRec TmeCpl SpcCpl	C3	–	+	+	–	–	–	+	+
NBlkSndNBlkRec TmeCpl SpcCpl	C4	–	–	–	–	–	–	+	+
BlkSndBlkRec TmeDcpl SpcCpl	C5	–	–	–	–	+	–	–	+
NBlkSndBlkRec TmeDcpl SpcCpl	C6	–	–	–	–	+/–	–	–	+
BlkSndNBlkRec TmeDcpl SpcCpl	C7	–	+/–	+	–	+	–	–	+
NBlkSndNBlkRec TmeDcpl SpcCpl	C8	–	–	–	–	+/–	–	–	+
BlkSndBlkRec TmeCpl SpcDcpl(Ch)	C9	–	–	–	–	–	–	–	+
NBlkSndBlkRec TmeCpl SpcDcpl(Ch)	C10	–	–	–	–	–	–	–	+
BlkSndNBlkRec TmeCpl SpcDcpl(Ch)	C11	–	–	+	–	–	–	–	+
NBlkSndNBlkRec TmeCpl SpcDcpl(Ch)	C12	–	–	–	–	–	–	–	+
BlkSndBlkRec TmeDcpl SpcDcpl(Ch)	C13	+	–	–	+	+	+	–	+
NBlkSndBlkRec TmeDcpl SpcDcpl(Ch)	C14	–	–	–	+/–	+/–	+/–	–	+
BlkSndNBlkRec TmeDcpl SpcDcpl(Ch)	C15	+	–	+	+	+	+	–	+
NBlkSndNBlkRec TmeDcpl SpcDcpl(Ch)	C16	–	–	–	+/–	+/–	+/–	–	+
BlkSndBlkRec TmeCpl SpcDcpl(Tpc)	C17	–	–	–	–	–	–	+/–	+
NBlkSndBlkRec TmeCpl SpcDcpl(Tpc)	C18	–	–	–	–	–	–	+/–	+
BlkSndNBlkRec TmeCpl SpcDcpl(Tpc)	C19	–	–	+/–	–	–	–	+/–	+
NBlkSndNBlkRec TmeCpl SpcDcpl(Tpc)	C20	–	–	–	–	–	–	+/–	+
BlkSndBlkRec TmeDcpl SpcDcpl(Tpc)	C21	+/–	–	–	+	+	+/–	–	+
NBlkSndBlkRec TmeDcpl SpcDcpl(Tpc)	C22	–	–	–	+/–	+/–	–	–	+
BlkSndNBlkRec TmeDcpl SpcDcpl(Tpc)	C23	+/–	–	+/–	+	+	+/–	–	+
NBlkSndNBlkRec TmeDcpl SpcDcpl(Tpc)	C24	–	–	–	+/–	+/–	–	–	+
Multicast Patterns									
Multicast/Scatter	M1	–	–	–	–	+	+	+	+
JoinMsgs/Gather	M2	–	–	–	–	–	–	+	+
Response Patterns									
PreprocessAck	R1	–	–	–	+	+	–	+	+
PostProcessAck	R2	–	+	+	+/–	+/–	–	–	+/–
PostProcessResp	R3	–	+	+	+/–	+/–	–	–	+/–
PostProcessFault	R4	–	+	+	–	–	–	–	+/–
Blocking Receive Ack	R5	–	+	+	+	+	–	–	+
Non Blocking Receive Ack	R6	–	–	+	–	–	–	–	+

Table 1. Evaluation of selected middleware in terms of their support for the coupling integration patterns, multicast patterns (Section 5.1) and response patterns (Section 5.2).

7 JDecouple: An API for highly (de-)coupled systems

We have implemented JDecouple, a middleware API based on the concepts presented in this article. JDecouple exposes simple abstractions that enable applications to be integrated according to any of the 24 integration coupling patterns presented in Figures 13, 14, and 15. The amount of coding effort required to achieve any one of them is minimal. This degree of suitability and flexibility, we believe, is unique to JDecouple. Having said that, it is *possible* to implement all of these 24 patterns of integration using most middleware. However, for those patterns not directly supported by a particular solution, the effort required becomes non trivial (confer Table 1). Advantages of JDecouple include:

– The ability to provide direct support for a wider range of (de-)coupled types of interactions.

– The ability to provide this wide range of interaction styles through one, simple API.

JDecouple exposes these patterns, optionally, at the granularity of elementary interactions, (i.e. one message exchange between two endpoints). Thus it is possible, but not necessary, for a JDecouple endpoint to send messages using different coupling patterns. This is because the configuration of the interaction is determined by the method chosen and the arguments used. JDecouple is written in Java 5, and makes extensive use of its new language features, including 'generics', and the new 'concurrent' tool-set. This first prototype of JDecouple provides messaging between threads running in the same application. Therefore the implementation has not yet been tested against problems such as latency, unreliability, and lack of bandwidth.

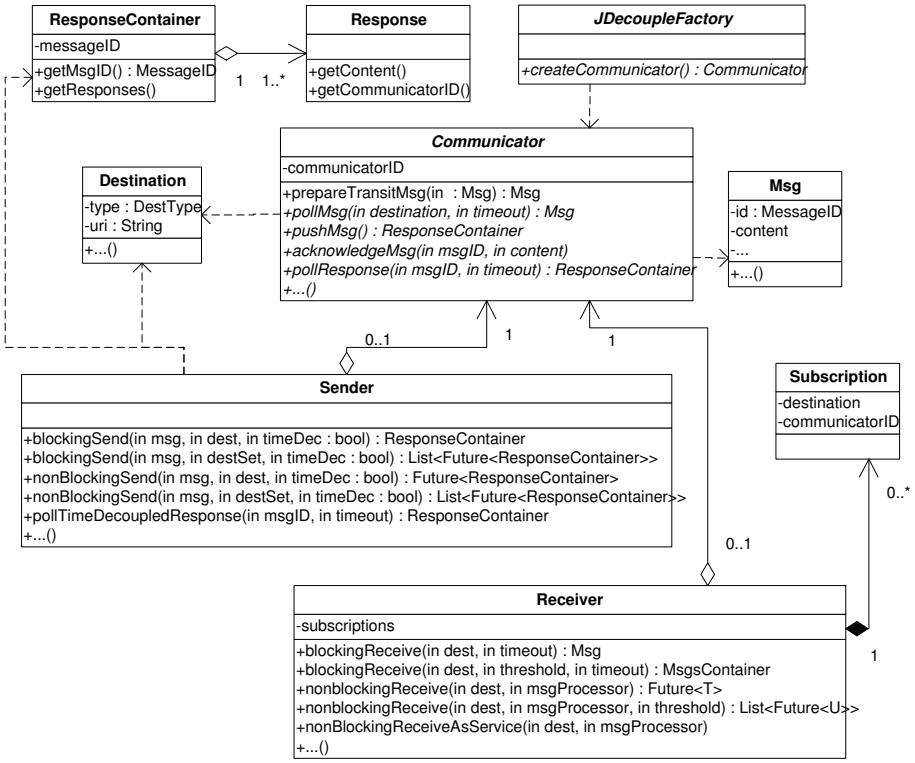


Fig. 19. UML diagram showing key classes of the JDecouple API.

Figure 19 shows that the abstract class `Communicator` is central to the JDecouple API. Subclasses of `Communicator` provide the necessary functionality, required by the `Sender` and `Receiver` classes, in order to perform all forms of

interactions between remote applications. For instance, an MSMQ implementation would enable interactions coded over the JDecouple API, to be executed over the MSMQ transport layer. The `JDecoupleFactory` interface provides a dynamic means of creating these alternative communicators without having to rewrite code. For instance an implementation of this interface could create either a JMS, TCP, or SOAP/HTTP communicator based on the contents of a text-based configuration file. More information about JDecouple can be obtained from `http://sky.fit.qut.edu.au/~aldredl`.

Integration coupling pattern C13. The following listings demonstrate the implementation of coupling pattern C13 (blocking-send, blocking-receive, time-decoupled, space-decoupled-channel) as presented in Figure 14. This is a classic messaging pattern supported by most Message Oriented Middleware.

Listing. 1. Performs a time-decoupled, space-decoupled, blocking-send.

```
1     ...
2     JDecoupleFactory JDecoupleFactory = new LocalJDecoupleFactory();
3     try{
4         Communicator communicator = JDecoupleFactory.createCommunicator();
5         Sender sender = new Sender(communicator);
6         Channel channel = (Channel) communicator.lookup(CHANNEL_URI);
7
8         Message message = new Message();
9         message.setContent("Hello World");
10
11        sender.blockingSend(message, channel, TimeCoupling.decoupled);
12    } catch (JDecoupleException e) { ... }
```

In Listing 1, line 2 creates a factory capable of instantiating communicator objects for intra-application (in-memory) communication. Lines 5 - 6 create the sender and obtain a reference to the channel. Line 11 sends the message over this channel in a time-decoupled manner. Line 12 is needed because lines 4 and 11 can throw either a TransportException, NotFoundException, or a PermissionException (all sub-types of JDecoupleException).

Listing. 2. Performs a space-decoupled, blocking receive.

```
1     ...
2     try{
3         Communicator communicator = JDecoupleFactory.createCommunicator();
4         Receiver receiver = new Receiver(communicator);
5         Channel channel = (Channel) communicator.lookup(CHANNEL_URI);
6
7         Message message = receiver.blockingReceive(channel, Receiver.NEVER_TIMEOUT);
8
9         System.out.println("Received msg ID: " + message.getID());
10        System.out.println("Contains: " + message.getContent());
11    }
12    catch (JDecoupleException e) { ... }
13    catch (TimeoutException e) { ... }
```

In Listing 2 lines 3 - 5 create the receiver, and obtain a reference to the same channel. Line 7 performs the receive - returning a `Message` object. The output from executing Listings 1 and 2 is: `Received msg ID: M-9223372036854775808&C1365919705802591056 Contains: Hello World`

Inter-library loan query service. Listings 3 and 4 present a scenario where a library, upon receiving a request to borrow a book that it does not have, checks with its nearby partner libraries if they have a copy of the book.

One library broadcasts the customer request to all libraries. Each library performs an internal search for books meeting the requested criteria, and returns a message containing the results of its search. The original library, upon receiving the responses from each partner, filters out all but the most favourable responses and presents them to the library customer.

This broadcast of many copies of a message, their receipt/response, followed by the filtering of all responses strongly resembles one of the more advanced patterns of enterprise integration – the *scatter gather* pattern [14]. The authors of this pattern propose a solution using a callback over a return address. This approach decouples the endpoints to a great degree, however such an approach typically requires significant effort to build receivers able to parse the incoming message for a return address, and then use that return address to obtain a channel/address object over which to start a new interaction – carrying the response.

A simpler solution to the scatter gather pattern would remove the need for each subscriber to explicitly call back the publisher, alleviating the need for a dedicated responses channel, and the use of a return address in the published message.

Using JDecouple, a non-blocking-send can be published onto a topic (space-decoupled-topic), and when the results are ready, the publisher may rendezvous with the responses. A non-blocking-receive will enable receiver endpoints to contain business logic that is automatically invoked by JDecouple, alleviating the need to explicitly handle requests and manage threads. A time-coupled interaction style will allow the library customers to know the results of their search in "real-time", all at once. Therefore, the most suitable coupling integration pattern seems to be C20 (non-blocking-send, non-blocking-receive, time-coupled, space-decoupled-topic, see Figure 15). By exploiting JDecouple's support for "postprocess acknowledgement" (pattern R3) a suitable solution to this integration problem is possible.

Listing. 3. Publishes a search request to all libraries, and filters for the best responses.

```
1     ...
2     try {
3         JDecoupleFactory JDecoupleFactory = new LocalJDecoupleFactory();
4         Communicator communicator = JDecoupleFactory.createCommunicator();
5         Sender sender = new Sender(communicator);
6         Topic topic = (Topic) communicator.lookup(TOPIC_URI);
```

```
7
8         Message message = new Message();
9         message.setContent(BOOK_REQUEST);
10
11        Future<ResponseContainer> futureResponses =
12                sender.nonBlockingSend(message, topic, TimeCoupling.coupled);
13
14        updateCustomerUI();
15
16        ResponseContainer responseContainer = futureResponses.get();
17        List<Response> subscriberResponses = responseContainer.getResponses();
18        List<Response> goodResponses = filterResponses(subscriberResponses);
19        ...
20    } catch (JDecoupleException e) { ... }
21    catch (ExecutionException e) { ... }
22    catch (InterruptedException e) { ... }
```

In Listing 3 lines 3 - 6 deal with instantiating the message sender and looking up the topic to which to send the message. Line 11 deals with publication of the book request to all affiliated libraries. Line 14 updates a user interface, informing the customer that the search is underway (not JDecouple code). Now that customer UI is updated we rendezvous with the responses of the prior send at line 16 – retrieving an object containing responses from each subscriber. Line 18 (not JDecouple code) filters the libraries' responses for favourable ones. Line 20 catches any JDecoupleExceptions. Calling `get()` on a `java.util.concurrent.Future` object (line 16) may throw an ExecutionException (possibly due to a message processing fault at the remote endpoint) or an InterruptedException - hence lines 21 and 22.

Intrinsic to our inter-library loan example, and to the scatter gather pattern, is that the content of each of the many response messages is derived from some application logic. JDecouple provides an interface containing two generic methods:

```
public interface MessageProcessor<V>{
    public V processMessage(Message message) throws Exception;
    public <U extends Serializable>U getResponse();
}
```

Implementations of `MessageProcessor` should provide the application logic needed to process the message, and (optionally) to format a response. The object returned from `processMessage` is made available to the receiver, whereas the object returned from `getResponse` gets put onto the JDecouple message bus, and optionally returned to the sender.

Listing. 4. Creates a search receive/response server for any partner library.

```
1     ...
2     try {
3         JDecoupleFactory JDecoupleFactory = new LocalJDecoupleFactory();
4         BookRequestProcessor bookRequestProcessor = new BookRequestProcessor();
5
```

```
6         Communicator receiveCommunicator = JDecoupleFactory.createCommunicator();
7         Receiver receiver = new Receiver(receiveCommunicator);
8         Topic topic = (Topic) receiveCommunicator.lookup(TOPIC_URI);
9         receiver.subscribe(topic, TOPIC_PASSWORD);
10
11        receiver.nonBlockingReceiveAsService(topic, bookRequestProcessor);
12    } catch (JDecoupleException e) { ... }
```

In Listing 4, line 4 instantiates a custom implementation of the `MessageProcessor` interface. Lines 6 - 9 create the receiver, obtain the topic reference, and subscribe the receiver to the topic. Line 11, registers the receiver, and in turn the `MessageProcessor` with JDecouple. JDecouple may now invoke `processMessage` of the `BookRequestProcessor` object using an event-based approach (similar to `onMessage` in the JMS).

In JDecouple each type of destination (i.e. address, channel, or topic) are capable of queueing their incoming messages. Consequently we can guarantee that the order in which messages arrive on the bus is the order in which they are consumed. Therefore, if the sender and receiver on one destination are blocking, JDecouple can guarantee preservation of message sequence. Another feature of JDecouple is its ability to perform multicast, and multiple message joining.

8 Related Work

Cypher and Leu [9] provided a formal semantics of blocking/non-blocking send/receive which is strongly related to our work. Their primitives were defined in a formal manner and related to the MPI [23]. This work does not consider space decoupling. Our research adopts concepts from this work and applies them to state of the art middleware systems. Our research differs by combining synchronisation concepts with the principles of time and space decoupling (originating from Linda [11]). Our work is also unique in its unifying effect over these dimensions, and the way coupling integration primitives may be used as a basis for middleware comparison.

Charron-Bost, Mattern, and Tel [6] provide a highly theoretical and insightful formalisation of synchronous, asynchronous, and causally ordered communication. This study introduces a notion of generalisation among these forms of communication according to sequences of messages at the global perspective and cyclic dependencies between them. They propose an increasing gradation of strictness starting with asynchronous computation (akin to all forms of time-decoupled communication), through FIFO computations (akin to message sequence preservation), through causally ordered computations, and finally to the most strict form - synchronous computations (akin to synchronisation-coupled, time-coupled communication).

Cross and Schmidt [8] discussed a pattern for standardising quality of service control for long-lived, distributed real-time and embedded applications. This was a proposal for "configuration tools that assist system builders in selecting compatible sets of infrastructure components that implement required services". In the context of that paper no proposals or solutions were made for this, however

the proposals of our article perhaps provide a fundamental basis for the selection of compatible sets of infrastructure.

Thompson [25] described a technique for selecting middleware based on its communication characteristics. Primary criteria include blocking versus non-blocking transfer. In this work several categories of middleware are distinguished, including conversational, request-reply, messaging, and publish-subscribe. The work, while insightful and relevant, does not attempt to provide a precise definition of the identified categories and fails to recognise subtle differences with respect to non-blocking communication.

Schantz and Schmidt [22] described four classes of middleware: *Host infrastructure middleware* (e.g. sockets), *Distribution middleware* (e.g. CORBA [12], and RMI [16]), *Common Middleware Services* (e.g. CORBA and EJB), and *Domain Specific Middleware Services* (e.g. EDI and SWIFT). This classification provides a compelling high-level view on the space of available middleware, but it does not give a precise indication of the subtle differences between alternatives in the light of architectural requirements.

Tanenbaum and Van Steen [24] described the key principles of distributed systems. Detailed issues were discussed such as (un-)marshalling, platform heterogeneity, and security. The work was grounded in selected middleware implementations including RPC, CORBA, and the World Wide Web. Our work is far more focussed on coupling at the architectural level, and therefore complements the more detailed issues provided by Tanenbaum and Van Steen.

Barros et. al. [4] produced a set of interaction patterns. The paper is oriented towards Web services and their relationship with well known technologies such as Business Execution Language for Web Services (BPEL4WS). Our article is distinguished from this work in that the patterns we present are less ad hoc, and the formality of our approach has led to new insights into a suitable abstraction set for integration. Also, our approach focuses on basic interactions (i.e. one-way and request-response) as opposed to compositions of interactions.

The workflow patterns [1] is an effort to capture a set of patterns over the control flow perspective of workflow. The topic of this article is different and the method is more technical, however both works aim to find insights into their respective domains through seeking suitable abstractions.

This article is an extended version of [2]. In addition to many refinements, this article adds publish-subscribe, responses, fault propagation, multicast, and message joining to the previous work on the coupling dimensions. This article also introduces a newly developed prototype (JDecouple) that performs integration according to the semantics presented in the CPNs.

9 Conclusion

This article has presented a set of formally defined notational elements to capture architectural requirements with respect to coupling. The proposed notational elements are derived from an analysis of middleware in terms of three orthogonal dimensions: space time and synchronisation. This analysis goes beyond previous middleware classifications by identifying certain subtleties with respect to time coupling. In previous middleware analyses, when two endpoints are coupled in

time, they are generally considered to be synchronous, and in the reverse case they are considered to be asynchronous (e.g. [14]). However, such an imprecise definition does not provide any differentiation between sockets and RPC, which are both time-coupled. Clearly there is more to coupling than the general consensus that time-coupled interactions are always synchronous. We consider the terms 'synchronous' and 'asynchronous' too imprecise to constitute a foundation for defining models of integration. The twenty-four coupling integration patterns identified in this article do provide a precise, and conceptually suitable palette for defining the coupling aspect of integration models. They also offer usefulness as a requirement set used in selecting middleware solutions suitable for a particular integration scenario. This set of patterns is unique in that it unifies and organises existing knowledge in the domain of integration coupling.

To demonstrate the feasibility of supporting all the coupling patterns identified in this article, we have defined and implemented an API, namely JDecouple, in which any of the patterns can be achieved using a small, unified set of primitives. Using JDecouple, it is also possible to easily implement interactions ranging from asynchronous (synchronisation-decoupled, time-decoupled) messaging through to the Scatter-Gather pattern of [14]. The current JDecouple prototype is only a proof-of-concept, and operates in-memory, as opposed to using distributed communication. We expect future versions of JDecouple to rely on JMS, TCP, and/or XML/HTTP as a transportation layer.

Disclaimer The assessments we made of middleware products and standards with respect to the coupling integration patterns are based on the tool or standard documentation. They are true and correct to the best of our knowledge.

Acknowledgement This work is partly funded by an Australian Research Council Discovery Grant "Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages". The third author is funded by a Queensland Smart State Fellowship.

References

1. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
2. 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.
3. Apache axis homepage. `http://ws.apache.org/axis/` accessed May 2006.
4. A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns. In *In Proceedings of the 3rd International Conference on Business Process Management (BPM), Nancy, France, September 2005*, pages 302–318. Springer Verlag, 2005.
5. 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.

6. B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996.
7. CPN Tools homepage. `http://wiki.daimi.au.dk/cpntools/` accessed June 2006.
8. Joseph K. Cross and Douglas C. Schmidt. Applying the quality connector pattern to optimise distributed real-time and embedded applications. *Patterns and skeletons for parallel and distributed computing*, pages 209–235, 2003.
9. 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.
10. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
11. David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
12. Object Management Group. *Common Object Request Broker Architecture: Core Specification*, 3.0.3 edition, March 2004. `http://www.omg.org/cgi-bin/apps/doc?formal/04-03-01.pdf` accessed May 2006.
13. M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Haase. *Java Messaging Service API Tutorial and Reference*. Addison-Wesley, 2002.
14. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, MA, USA, 2003.
15. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
16. Sun Microsystems. Java remote method invocation specification. `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html` accessed June 2006.
17. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. MIT Press, Cambridge, USA, 1997.
18. Websphere MQ family. `http://www-306.ibm.com/software/integration/wmq/` accessed June 2006.
19. Microsoft Message Queuing. `http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx` accessed June 2006.
20. D. Quartel, L. Ferreira Pires, M. van Sinderen, H. Franken, and C. Vissers. On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems*, 29(4):413 – 436, 1997.
21. E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
22. R. Schantz and D. Schmidt. *Encyclopedia of Software Engineering*, chapter Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. Wiley & Sons, New York, USA, 2002.
23. M. Snir, S. Otto, D. Walker S. Huss-Lederman, and J. Dongarra. *MPI-The Complete Reference: The MPI Core*. MIT Press, second edition, 1998.
24. A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
25. J. Thompson. Toolbox: Avoiding a middleware muddle. *IEEE Software*, 14(6):92–98, 1997.

A Rationale behind the evaluation of Standards and Tools against the Patterns

Table 1 presented an evaluation of middleware standards and tools, in terms of their ability to directly support or partially support the proposed patterns of this article. First of all it is important to establish that with any of these standards and solutions, incorporating enough 'work-arounds' will support all of the patterns, so these judgements were made based upon the level of implementation effort required to achieve the pattern.

The patterns one through to twenty four are composed of more fundamental concepts. These twenty four patterns (the coupling integration patterns) can be grouped into several sets based on the more fundamental concepts they are composed from. This makes the evaluation simpler because if a tool or standard does not support one of these fundamental concept we can expect that it will not support any of the patterns composed form this concept. However, supporting a concept (e.g. non-blocking receive) does not imply support for all of the patterns composed from this concept.

Table 2 shows the relative levels of support by selected middleware solutions and standards for the dimensions of decoupling. Solutions that directly support a pattern were given a plus ('+'). Those able to support a pattern using more than a little effort were given a plus minus ('+/–'), and those requiring greater effort were assigned a minus ('–'). A '–' symbol, assigned to a standard/solution, does not mean that the achievement of this pattern is impossible; rather, the 'work-arounds' necessary to achieve the pattern, using this standard/solution, appear too complex.

Dimension	**Option**	#	Java Spaces	Axis	CORBA	JMS	Websphere MQ	MSMQ	MPI	JDecouple
Sync.	Blocking Send	1	+	+	+	+	+	+	+	+
	Non Blocking Send	2	–	–	–	+/–	+/–	+/–	+	+
	Blocking Receive	3	+	–	–	+	+	+	+	+
	Non Blocking Receive	4	+	+	+	+	+	+	+	+
Time	Time Coupled	5	–	+	+	–	–	–	+	+
	Time Decoupled	6	+	+/–	+	+	+	+	–	+
Space	Space Coupled	7	–	+	+	–	+	–	+	+
	Space Decoupled – Channel	8	+	–	+	+	+	+	–	+
	Space Decoupled – Topic	9	+/–	–	+/–	+	+	+/–	+/–	+

Table 2. Support of selected middleware solutions and standards for the dimensions of (de-)coupling

A.1 Java Spaces

Java Spaces supports a blocking-send through its `write` operation. However it does not support a non-blocking send, hence it was given a '–' in Table 2. Consequently, in Table 1, all the patterns composed from non-blocking send were assigned a '–' (namely patterns C2, C4, C6, C8, C9, C10, C12, C14, C16, C18, C20, C22, and C24).

Java Spaces supports blocking-receive through its `read` or `take` operations, and supports non-blocking receive through its `notify` operation, hence options '3' and '4' were assigned a '+' in Table 2.

Java Spaces supports space-decoupling over a channel, but does not directly support space-coupling, as shown in Table 2. Consequently, patterns C1 – C8 of Table 1 were assigned a '–'.

Space-decoupling over a topic (e.g. publish-subscribe) is partially achieved if each 'subscriber' uses a `read` operation. Each receiver gets a copy of the message because the call to `read` does not remove the message from the space. Hopefully the message will be removed from the space before any receiver reads the same message twice. A simple, but not fail-safe 'work-around' to prevent this problem is to write the message to the space with a very short lease.

Java Spaces, is time-decoupled, and is not time-coupled. Consequently patterns C1 – C4, C9 – C12, and C17 – C20 were assigned a '–' in Table 1.

Java Spaces does not provide primitives for sending messages over an arbitrary set of templates, and we therefore rule out multicast. We rule out support for message joining because it does not support its `take`, or `read` operations over arbitrary sets of templates.

Java Spaces does not directly support responses, therefore patterns R1 – R6 of Table 1 were assigned a '–'.

A.2 Axis

Axis is a SOAP engine that primarily uses HTTP as transport. Like HTTP, it only offers a blocking-send, and a non-blocking-receive. Despite the fact that Axis can be configured to use JMS as a message transport service, it does not expose a non-blocking send or a blocking-receive in its API. Non-blocking send and blocking receive were assigned a '–' in Table 2. Consequently all the patterns composed from these were assigned '–' in Table 1.

Axis, supports time-coupling but only when layered over a JMS implementation could it possibly support time-decoupling.

Axis directly supports space-coupling, but does not support space-decoupling. Despite the fact that JMS supports space-decoupling over channels and topics Axis does not expose constructs that allow users to exploit either of these features. Consequently, in Table 1, patterns C9 – C24 were assigned '–'.

Axis provides no direct support for multicast, or for message joining.

Axis supports R2 – R5 of the response patterns directly. Preprocess acknowledgement (R1) is not supported. A 'work-around' solution would require forking

off a thread in the server to process the message while sending a receipt acknowledgement in the response from the main thread. Non-blocking receive of responses is not possible without using callbacks, hence it is assigned '–' in Table 1.

A.3 CORBA

CORBA supports a blocking-send because the CORBA client blocks on remote object calls ,at least until the request has reached the ORB. It supports non-blocking-receive because the remote object servicing requests never makes an explicit request to wait for an incoming request, this gets managed by the ORB. Nevertheless, CORBA provides no direct support for a non-blocking-send or a blocking-receive operation.

CORBA traditionally offers a time-coupled means of interacting, and using what it refers to as an 'asynchronous' style it provides direct support for time-decoupled interactions.

CORBA's naming service is the primary way to address remote objects. The naming service maps a name to one remote object, as opposed to one of many, hence CORBA directly supports space-coupling. CORBA allows clients to obtain remote object references using Interoperable Object Group Reference (IOGR). This sort of remote object reference refers to one of many object implementations, and therefore CORBA supports space-decoupling over a channel. CORBA also has support for publish-subscribe, but achieving this style of interaction is not as straightforward as it should be, and therefore we consider that it only partially supports this (see Table 2).

CORBA does not directly support multicast or message joining hence we gave it '–' for M1 and M2 of Table 1.

CORBA would require the use of 'work-arounds' to provide the client with an acknowledgement before the request gets processed, therefore we gave response pattern R1 a '–' in Table 1. Otherwise CORBA fully supports all of the other response patterns better than any other solution/standard.

A.4 Java Message Service

The JMS standard directly supports blocking-send, blocking-receive, and non-blocking receive. However, it does not directly support non-blocking send. Nevertheless, most implementations of JMS can be configured with a sender endpoint, and middleware service on the same machine. Blocking-send swaps the message from the endpoint to the middleware service and the sending endpoint does not have to block while the message is being transferred over the wire. Hence we assign JMS a '+/–' for Option '2' in Table 2. Consequently all of the coupling integration patterns composed from non-blocking send in Table 1 (e.g. C14, C16, C22, and C24) are assigned, at best, a '+/–'.

JMS, being a MOM driven standard, natively supports time-decoupling but not time-coupling. Consequently all patterns composed from time-coupling in Table 1 (C1 – C4, C9 – C12, and C17 – C20) were assigned a '–'.

JMS supports both forms of space-decoupling (channel and topic), but not space-coupling. Consequently all patterns in Table 1 composed from space-coupling (C1 – C8) were assigned a '–'.

JMS, however does not support either of the multicast patterns.

JMS is not a request-response driven standard, but despite this it supports preprocess acknowledgements (R1) directly via session acknowledgements. It supports blocking-receive acknowledgements (R5) directly through its `QueueRequestor` and `TopicRequestor` classes. Post-process acknowledgements (R2) and post-process responses (R3) are supported through the same classes. However, we deem this support only partial ('+/–') due to the need to parse for a return address and begin a new interaction at the responder (as discussed in Section 5.2).

A.5 Websphere MQ

Websphere MQ is the MOM component of the Websphere suite, by IBM. Websphere MQ provides an implementation of the JMS standard and, of course, supports every pattern that JMS covers.

Additionally, Websphere MQ allows the configuration of one particular endpoint to exclusively receive messages off a channel. Therefore Websphere MQ fully supports space-coupling. Consequently the time-decoupled, space-coupled patterns (C5 – C8) of Table 1 were assigned either '+', or '+/–' according to Websphere MQ's support for synchronisation decoupling.

A.6 MSMQ

MSMQ is the MOM implementation by Microsoft. It provides MOM support to the BizTalk process application server and to the new Windows Communication Foundation.

MSMQ directly supports blocking-send, blocking-receive, and non-blocking receive. However, like the JMS, it does not directly support non-blocking send. Nevertheless the same work-around to achieve non-blocking send for JMS can be performed using MSMQ. Hence we assign MSMQ a '+/–' for Option '2' in Table 2. Consequently all of the coupling integration patterns composed from non-blocking send in Table 1 were assigned at best a '+/–'; and possibly lower, depending on the support for other dimensions.

MSMQ, like most MOM solutions does not support time-coupling. Hence we assigned a '–' to all patterns in Table 1 (C1 – C4, C9 – C12, C17 – C20) composed from time-coupling.

MSMQ has full support for space-decoupling over a channel but would require non trivial 'work-arounds' to achieve space-coupling. Therefore we assigned MSMQ a '–' for all the coupling integration patterns composed from space-coupling (C1 – C8).

With respect to *space-decoupling over a topic* MSMQ offers a `peek` operation in its API – allowing receiver endpoints to peek at messages in the queue. Using

`peek` to support space-decoupling over a topic is a work-around, in much the same class as the work-around required to support space-decoupling over a topic for Java spaces. Therefore, the rating we gave to those patterns composed from *space-decoupling over a topic* was at best a '+/–'.

MSMQ, to our knowledge, does not support multicast, or response patterns.

A.7 MPI

The Message Passing Interface, developed by a consortium of leading IT vendors and select members of the research community, was designed to enable parallel and distributed systems to exchange messages effectively. Using MPI, parallel applications are able to exploit processing on multiple CPUs for example, because the API is extremely efficient in its use of memory and the CPU.

MPI fully supports all forms of synchronisation (de-)coupling, providing explicit operations for blocking-send, non-blocking send, blocking-receive, and non-blocking receive – as is shown in Table 2.

MPI does not support time-decoupling – as shown in Table 2. Therefore all patterns in Table 1 (C5 – C8, C13 – C16, C21 – C24) composed from time-decoupling were assigned a '–'.

MPI does not support space-decoupling over a channel. Hence we rated patterns C9 – C16 in Table 1 with a '–'.

MPI is able to notify all members of a group with copies of the same message. This behaviour is strongly related to *space-decoupling over a topic*, however these groups are determined during build-time, or design-time. There seems to be limited support for joining a group at runtime, and no support for joining more than one group. We therefore rated MPI with at best a '+/–' for all patterns composed from *space-decoupled over a topic*.

MPI fully supports multicast, message joining and preprocess acknowledgement.

A.8 JDecouple

JDecouple is the experimental prototype of a new middleware API proposed as part of this article. It offers a full set of communication abstractions as part of its API that make it simple to integrate using any form of (de-)coupling, to Java developers. Additionally JDecouple fully supports both forms of multicast, and provides support for all six response patterns.