Dimensions of Coupling in Middleware

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

[1] Faculty of IT, 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. It is well accepted that different types of distributed architectures require different degrees of coupling. For example, in client-server and three-tier architectures, application components are generally tightly coupled, both with one-another and with the underlying middleware. Meanwhile, in off-line transaction processing, grid computing and mobile applications, 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 degree 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 to unambiguously communicate architectural requirements with respect to (de-)coupling. This article addresses this gap by: (i) formally defining three dimensions of (de-)coupling; (ii) relating these dimensions to existing middleware; and (iii) proposing notational elements to represent various coupling integration patterns. This article also discusses a prototype that demonstrates the feasibility of its implementation.

Keywords: Distributed architecture, Coupling, Decoupling, Asynchronous/synchronous communication, Message-Oriented Middleware

1 Introduction

Modern approaches to integrating distributed systems almost invariably rely on middleware. These middleware take many forms, ranging from distributed object brokers, to message-oriented middleware and enterprise service buses. Each family of middleware is designed to provide proven solutions to a certain set of distributed system integration issues, but how can one compare their relative strengths and weaknesses when they are so different at the core? Indeed, a unified 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 [6]. This lack of consensus can be observed even for the most elementary communication primitives, namely send and receive. For example in sending a message, the semantics of what happens (a) if it doesn't arrive, or (b) if it gets buffered (and if so if this happens locally/remotely); is usually not clear until the choice a technology has been made. 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" [10]. Coupling, or the way in which endpoints are connected, seems to exist at the very epicenter of this issue.

The style of coupling can vary from one family of middleware to another. For instance solutions inspired by CORBA [13] are based on a Remote Procedure Call (RPC) paradigm, wherein the sender and receiver applications typically become synchronised at the thread level. On the other hand Message-Oriented Middleware (MOM) solutions generally adopt the opposite approach citing an RPC style of approach as being inferior. In spite of these differences commercial middleware from each family (RPC, or MOM) usually support the others' style of coupling. For example the Java Message Service (JMS) [14] supports an RPC style.[1] But this only serves to add confusion to the choice of a middleware solution, and the lack of a formal overarching framework, or unifying theory here leads to all sorts of exceptions when working at the detailed level.

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 dimensions of coupling in terms of Coloured Petri Nets (CPNs) [16].
– A collection of *notational elements* for integration modelling based on the notions of coupling previously identified. These notational elements are given a visual syntax extending that of Message Sequence Charts (MSCs) [22].
– 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 *prototype* demonstrating the possibility of supporting all styles of coupling in a consistent and uniform way.

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 *uni-directional coupling integration patterns* that provide a basis for integration modelling. Section 5 introduces bi-directional interactions and incorporates them into the same conceptual framework. Section 6 presents a framework for comparing middleware solutions based on the uni-directional, and bi-directional patterns. Section 7 introduces *JCoupling*: an open source prototype that combines all of the proposals of this article. Sections 8 and 9 present related work and conclusions respectively.

This article is an extension of our previous analysis of coupling for elementary interactions [2]. It extends [2] to incorporate publish-subscribe techniques into the models of interaction. It also extends the previous work by encompassing more complex interactions such as request-response. The conceptual models also attempt to become more grounded, dealing remote fault reporting, aggregate messaging, and multicast. This article also introduces a new proof of concept prototype (JCoupling) that implements all the theoretical models presented in this article and in [2].

2 Background

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

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 (e.g. a message ID, timestamp, etc.) and/or data (also called the payload). It may be transactional, reliable, and may be transported over a “channel”, or even a TCP socket.

An *endpoint* is a communicating entity and is able to perform 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 communicate and process.

[1] The JMS TopicRequestor `http://java.sun.com/products/jms/javadoc-102a/javax/jms/TopicRequestor.html` (accessed April 2007).

An *interaction* refers to endpoints exchanging information [21]. The most basic interaction is a uni-directional message exchange (elementary interaction).

A *channel* is an abstraction of a message destination. This abstraction is similar to the notion of "queue" supported by JMS, WebsphereMQ [19], and Microsoft Message Queue (MSMQ) [20], but the crucial point here is that a channel is a logical place that many endpoints can share – each receiving and even competing for messages. This is opposed to an *address*, or a location that only one endpoint can receive from. Channels can be extended with many functions, including the preservation of message sequence [11, 10], authentication, and non-repudiation [15].

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.

Following Eugster's survey of Publish-Subscribe [11], three dimensions of decoupling can be identified:

– *Thread Decoupling* – (referred to by Eugster as 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.

The dimensions of decoupling have relevance to a large spectrum of middleware, including MOM, space-based [12], and RPC-based middleware.

3 Decoupling Dimensions of an Interaction

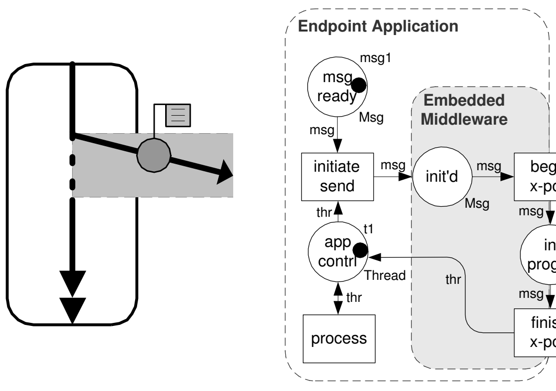
This section refines Eugster's dimensions of coupling and formalizes them using CPNs. CPNs were chosen for their ability to explicitly model state, parallelism, data, and for their formal semantics. All CPNs have been fully implemented and tested using CPN Tools [8].

3.1 The Thread-coupling Dimension

Thread-decoupling enables "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 thread decoupling as well as the CPN formalisation.

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 sender's thread 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.

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*



(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.

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 may be 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. A transition is enabled if *every one of its input places contains at least 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 specify the value of the token to be produced and may use arbitrary ML [18] functions. This feature will be used in Section 3.3.

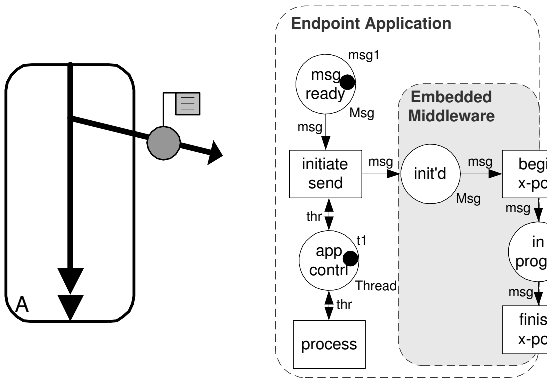
In Figure 1(b), when a message is ready (represented by a token inside the place "*msg-ready*") and the application is ready (represented by a token inside the place "*app-contrl*") the endpoint gives the message to the embedded middleware.[2] The endpoint yields its thread of control to the embedded 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, thread-coupled manner,

[2] In this series of CPNs we represent tokens as black dots. 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.

regardless of the behaviour exposed to the endpoint applications. Therefore this CPN may be "transition bounded" with remote systems.[3]

In a blocking send there is a thread 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. A *thread decoupling* is observable at the sender in the case of a *non-blocking send* [11]. A non-blocking send means that message transmission and local computation can be interleaved [24]. Figure 2(a) presents a notation for non-blocking send, based on the MSC notation [22]. 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 the embedded middleware of the application. Snir and Otto provide a detailed description of non-blocking send [24].



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

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

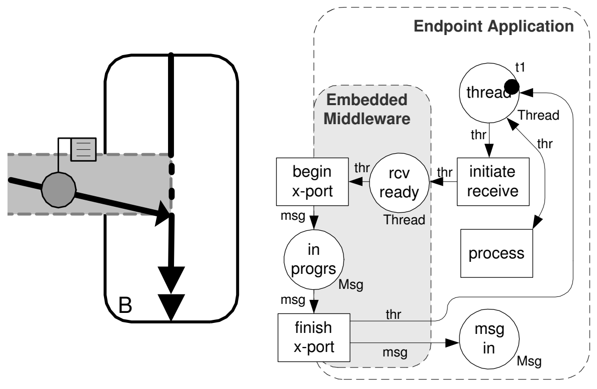
A non-blocking send is a necessary condition, but not a sufficient condition to achieve total thread decoupling, which is to say that the receive action must also be non-blocking. Rephrased, if both the send and receive are blocking (non-blocking) then a total thread coupling (decoupling) occurs. A partial thread 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" [19] 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 [10]. The definition of *blocking receive* is that the application must put a thread

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

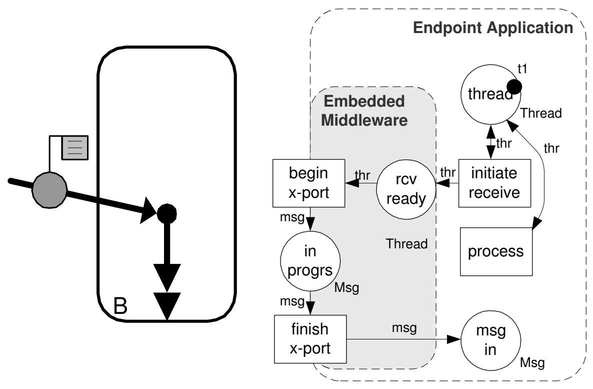
into a waiting state in order to receive the message (ownership of the thread is usually returned to the endpoint when the message is received). This means that the receiver thread is coupled to either the message sender directly or to some form of queue on a middleware service (depending on whether the middleware queues messages between the sender and receiver, or not). 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.

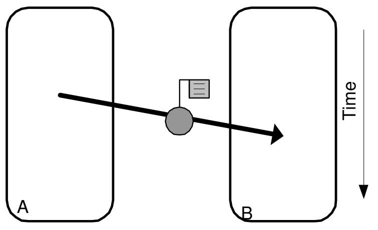
A well-known embodiment of non-blocking-receive is the event-based handler described in JMS [14]. Once a handler is registered with the middleware it is called-back when a message arrives. The Message Passing Interface (MPI) provides another embodiment of non-blocking receive that is not event-based [24], wherein the receiver polls for the presence of a new message.

Non-blocking receive, as a communication abstraction, seems less popular than blocking receive. This is probably because blocking receive interactions are simpler to program and debug [15]. One frequently observes statements about the merits of an asynchronous architecture. While such statements are indeed valid, they usually refer to the merits of queuing messages between the sender and the receiver, and have little to do with the fact that the threads of control are blocked or not. Therefore the threading dimension as presented in this section is orthogonal to the general usage of the word "asynchronous".

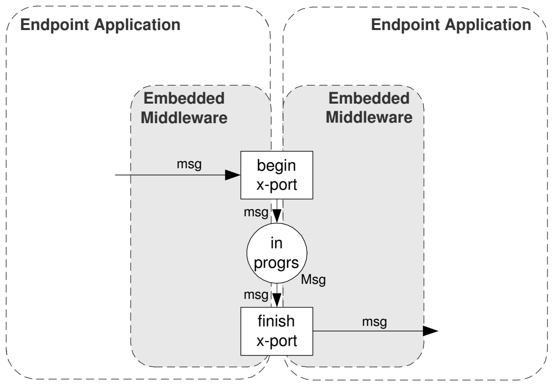
3.2 Time

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, like those typically found in MPI, cannot take place unless both endpoints are concurrently connected. There is no possibility of queueing messages between the two endpoints. In time-coupled arrangements the interaction begins with the message 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. This concept is presented in Figure 5 wherein the endpoint applications are joined directly at the bounding transitions ("*begin x-port*" and "*finish x-port*"). Time-coupled interactions accord with the general usage of the word "synchronous". Figure 5 does not show the endpoints. 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.

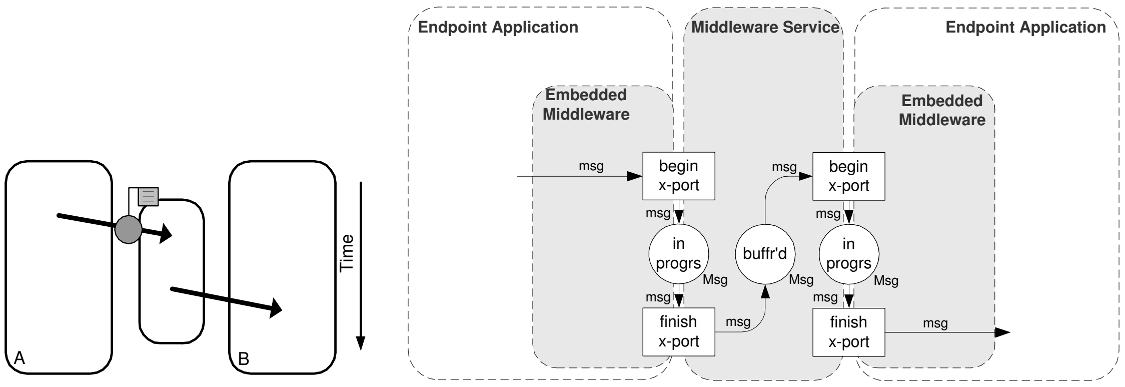
Fig. 5. *Time coupling* is characterised by transition-bounded systems.

Time-Decoupled. *Time-decoupled* interactions allow messages to be exchanged irrespective of whether or not each endpoint is concurrently operational. In this case messages *are queued* between the sender and the receiver. 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*".

MOM solutions such as Websphere MQ and MSMQ are typically 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. By "space" we refer a dimension of coupling that takes into account the degree to which sender endpoint can control which instance receives the



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

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

message. If the sender can totally control which endpoint instance would receive the message we can assume a high degree of space coupling, and vice versa.

Space Coupled. For an interaction to be *space coupled* the sender requires 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.

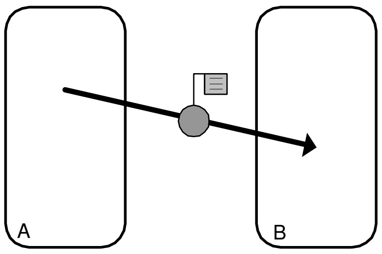
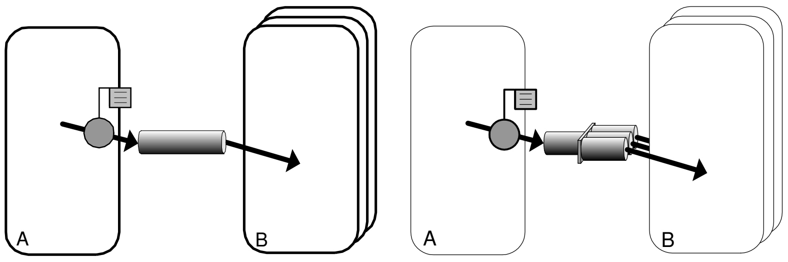


Fig. 7. *Notation, space coupled.* The sender directly addresses the receiver.

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 - one 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.

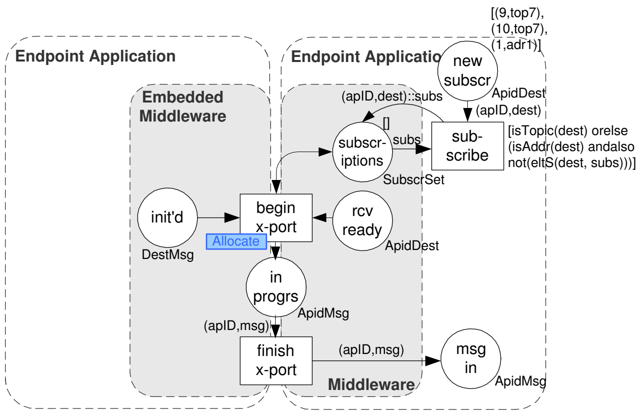


Fig. 9. CPN presenting a semantics for the space dimension. The transition *"begin x-port"* has been annotated with a box labelled *"Allocate"*, which means that this transition decomposes to a sub-net (see Figure 10).

In Figure 9, the transition *"begin x-port"* decomposes to the sub-net in Figure 10.[4] This captures the three options of the space dimension illustrated by figures 7, 8(a), and 8(b) and provides a suitable CPN representation. 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"*.

The CPN of Figure 9 presents the space dimension, and shows the boundary places ("*init'd*", "*rcv ready*", and "*in progrs*" etc.) which are common to the CPNs of other dimensions. By linking with these places a sender can bind any topic, channel or address to a message, pushing the token into the place *"init'd"* (which is typed `DestMsg`). Likewise, a receiver, linking to this model can push a token into the place *"rcv ready"*. This place is typed `ApidDest` and it identifies this receiver's application and the desired message source (i.e. destination).

Figure 10 presents the sub-net decomposition of the transition *"begin x-port"* (from Figure 9). This CPN shows how a unified underlying model *could* perform any of the three options within the space dimension (i.e. topic, address, or channel). From this we can conclude that it is possible to offer a clean implementation to all three forms of space (de-)coupling within the one middleware solution.

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). Note that a subscription is *essentially* a combination of a reference to a destination (of type `topic`) and a backward reference to the subscribing application. Similarly, an address *essentially* consists of a destination reference (of type `address`) being bound to one listening application. This similarity explains the fact that transition "*begin addr*" shares the input place "*subscriptions*" with transition "*begin topic*".[5]

[4] 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.

[5] Shown in Figure 9, is the transition for adding a new subscription to either an address or a topic. In this a transition guard prevents more than one application binding to an address at any time, while allowing many applications to subscribe to the same topic.

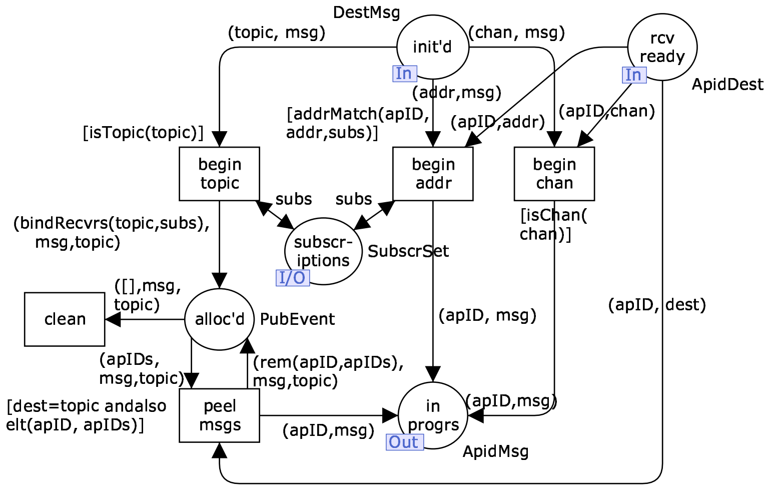


Fig. 10. This CPN is the 'Allocate' sub-net refining transition *"begin x-port"* in Figure 9.

Firing transitions "*begin addr*" or "*begin chan*" require the appropriately typed input tokens, and produce one token referring to the message and its intended application. The difference between the two is that "*begin addr*" can guarantee its token output will always identify the same application for any input address, whereas "*begin chan*" cannot guarantee which application "wins" the message. 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 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.

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 in 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, however they do not cut into the coupling dimensions.

3.4 Summary

Having conceptualized the three dimensions of decoupling formally has uses. Firstly, it can provide the opportunity to mathematically prove some properties about integrations. Secondly the concepts can clearly delineate fundamental differences between various forms of middleware, opening the potential for using the dimensions as a middleware selection instrument.

The dimensions of decoupling included "thread 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 Threading, Time, and Space

We contend that the dimensions of decoupling presented in the previous section are orthogonal to each other. Hence, any option (of the four) for thread-coupling can be combined with any option (of the two) for time-coupling, which in turn can be combined with any option (of the three) for space-coupling. Hence, for one-directional messaging, there are twenty four ($2^2 * 2 * 3 = 24$) possible ways to combine these three dimensions. We use the term *coupling integration pattern* to designate a given combination of options across the three dimensions. The semantics of each *coupling integration pattern* is precisely defined by a CPN, which is obtained by combining the corresponding CPNs for each of the three dimensions as explained below.

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.

CPN Merging. By combining the CPNs for each of the three dimensions, using CPN Tools, we have verified that the dimensions are indeed orthogonal. In other words, any CPN from each of the dimensions can be combined with any CPN from any other dimension, and the resulting merged CPN preserves the behaviour of its constituent CPNs.

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.

Minor adjustments are required when merging the CPNs. Due to space constraints we will only present the procedure we followed to create two of the twenty four possible merged CPNs. The other twenty two possible CPNs can be created similarly.

Figure 11 models a thread-coupled (i.e. 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 merge 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 intermediary between the blocking send and the blocking receive.

To produce Figure 11 we changed the types of selected places, and arcs to account for the space dimension. We specialised the type (from `Msg` to `DestMsg`) for the places *"msg ready"* and *"init'd"*, and likewise specialised the arc variables. We also specialised the type (from `Thread` to `ApidDest`) for the place *"rcv ready"* and inserted a new place *"msg req"*. A token in this place contains an identifier of the receiver (`Apid`) and the Channel/Topic/Address (`Dest`). The place labelled *"thread"* was specialised from `Thread` to `Appid`, and the places *"msg in"* and *"in progrs"* were specialised from `Msg` to `ApidMsg`. Finally, we added the sub-net *"Allocate"* (Figure 10) to the transition labelled *"begin x-port"* and linked the same transition to the place *subscriptions.*

A second example of a merged CPN is presented in Figure 12. It models a thread-decoupled, time-decoupled, space-decoupled (channel) interaction pattern. Being time-decoupled we

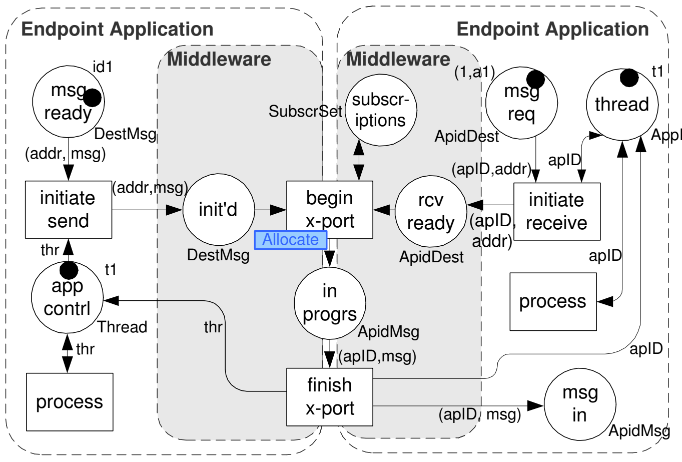


Fig. 11. Petri net of a thread-coupled, time-coupled, space-coupled interaction.

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.

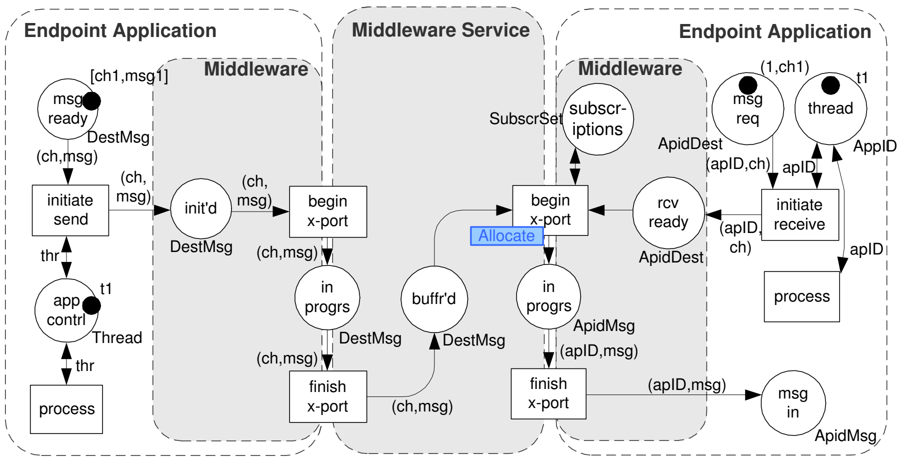


Fig. 12. CPN of a thread-decoupled, time-decoupled, and space-decoupled interaction.

In Figure 12 we specialised 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 specialised (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 coupling integration patterns, we combine the graphical notations from Section 3. Figures 13, 14, and 15 present these combined notations. These graphical notations are further extensions to MSCs [22], and are obtained by "overlaying" the notations for decoupling introduced in Section 3.

We contend that this set of coupling integration patterns can be used in defining interaction 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

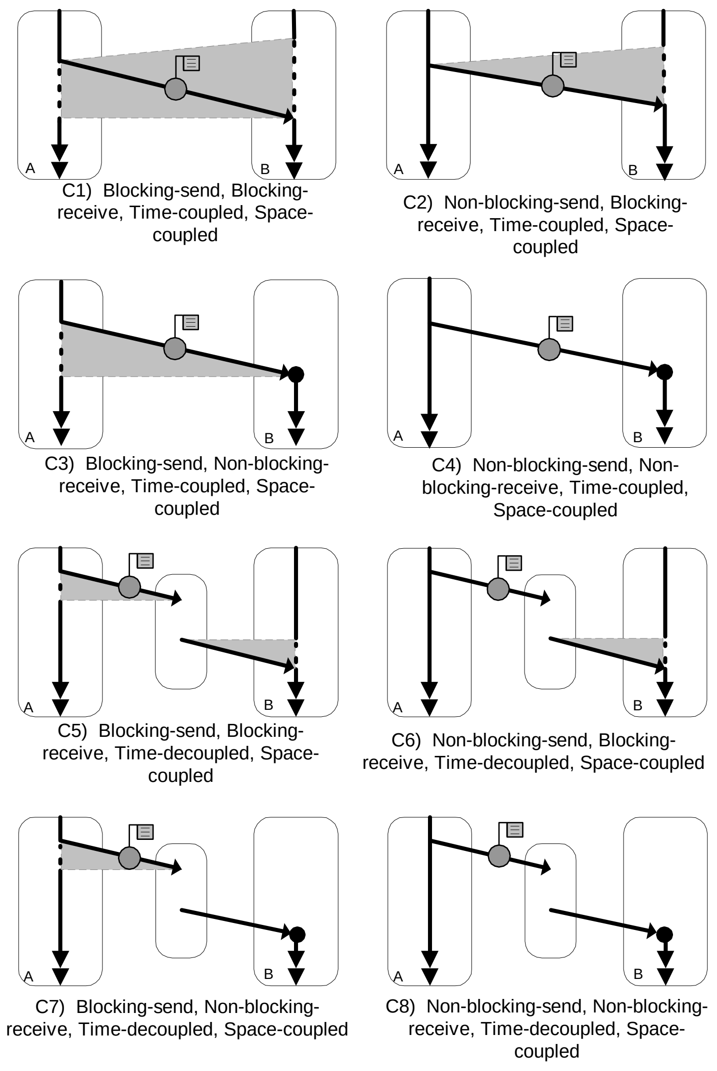

A
B
C1) Blocking-send, Blocking-receive, Time-coupled, Space-coupled
C2) Non-blocking-send, Blocking-receive, Time-coupled, Space-coupled
C3) Blocking-send, Non-blocking-receive, Time-coupled, Space-coupled
C4) Non-blocking-send, Non-blocking-receive, Time-coupled, Space-coupled
C5) Blocking-send, Blocking-receive, Time-decoupled, Space-coupled
C6) Non-blocking-send, Blocking-receive, Time-decoupled, Space-coupled
C7) Blocking-send, Non-blocking-receive, Time-decoupled, Space-coupled
C8) Non-blocking-send, Non-blocking-receive, Time-decoupled, Space-coupled

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

a given integration problem than others. For instance a multiplayer real-time strategy game would not have much use for time-decoupled interactions.

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 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.

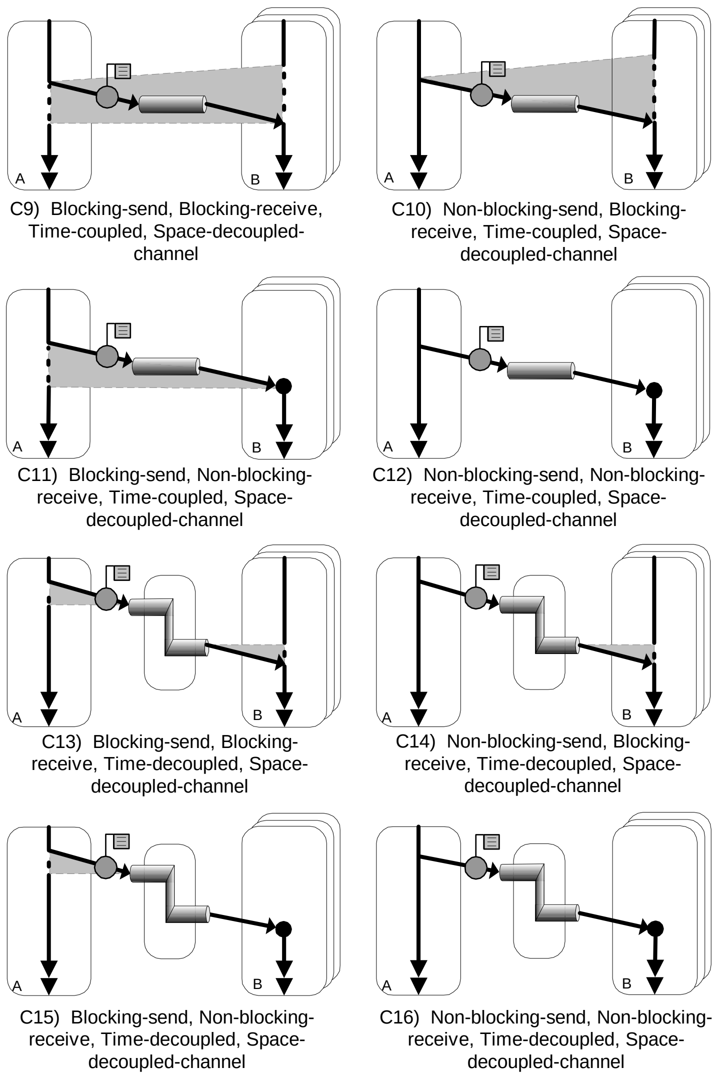

A
B
C9) Blocking-send, Blocking-receive, Time-coupled, Space-decoupled-channel
C10) Non-blocking-send, Blocking-receive, Time-coupled, Space-decoupled-channel
C11) Blocking-send, Non-blocking-receive, Time-coupled, Space-decoupled-channel
C12) Non-blocking-send, Non-blocking-receive, Time-coupled, Space-decoupled-channel
C13) Blocking-send, Blocking-receive, Time-decoupled, Space-decoupled-channel
C14) Non-blocking-send, Blocking-receive, Time-decoupled, Space-decoupled-channel
C15) Blocking-send, Non-blocking-receive, Time-decoupled, Space-decoupled-channel
C16) Non-blocking-send, Non-blocking-receive, Time-decoupled, Space-decoupled-channel

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

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 Bi-directional Interactions

Bi-directional interactions generally involve a requestor, and a respondent. As a general observation, middleware based on an RPC paradigm support bi-directional interactions, while Message-Oriented Middleware are oriented towards uni-directional messaging. For instance RPC-based middleware (e.g. CORBA [13]) supports "request-response" interactions, whereas MOM based middleware (e.g. Microsoft Message Queueing – MSMQ), do not natively support request-response interactions.

There are exceptions to this general observation, such as the JMS API. JMS provides partial support for request-response using its QueueRequestor and TopicRequestor classes, but these only serve to mask the two interactions occurring over the native JMS provider from the requestor's perspective. Further, these classes do not report remote exceptions and the TopicRequestor returns only the first response and ignores the responses of all other subscribers.

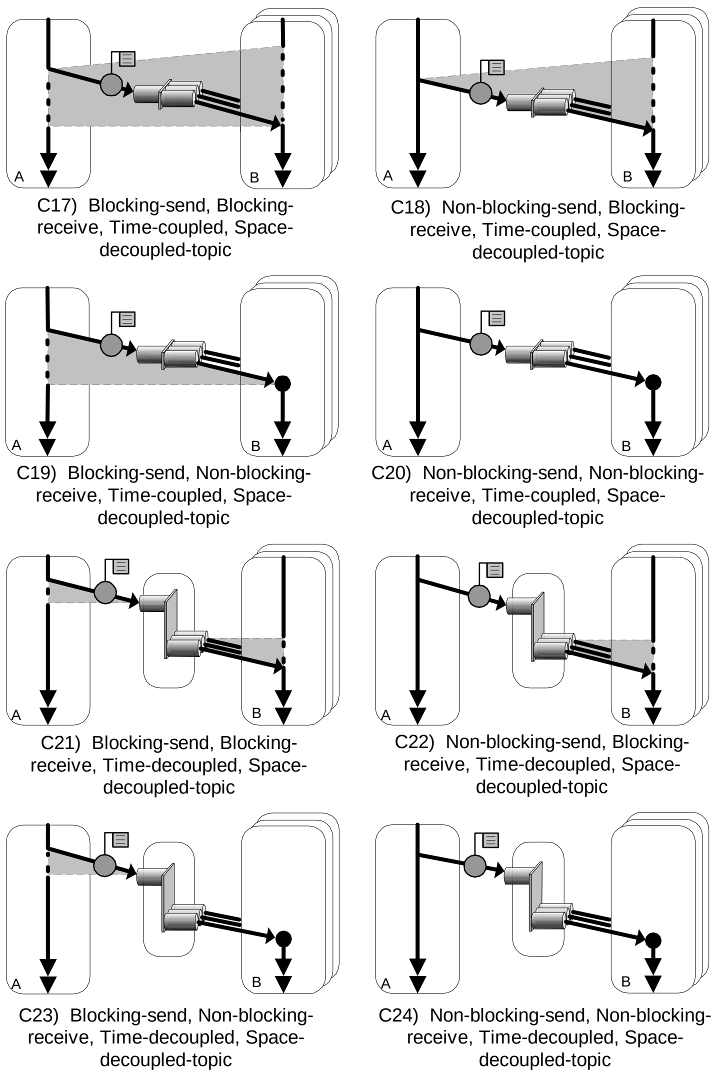

A
B
C17) Blocking-send, Blocking-receive, Time-coupled, Space-decoupled-topic
C18) Non-blocking-send, Blocking-receive, Time-coupled, Space-decoupled-topic
C19) Blocking-send, Non-blocking-receive, Time-coupled, Space-decoupled-topic
C20) Non-blocking-send, Non-blocking-receive, Time-coupled, Space-decoupled-topic
C21) Blocking-send, Blocking-receive, Time-decoupled, Space-decoupled-topic
C22) Non-blocking-send, Blocking-receive, Time-decoupled, Space-decoupled-topic
C23) Blocking-send, Non-blocking-receive, Time-decoupled, Space-decoupled-topic
C24) Non-blocking-send, Non-blocking-receive, Time-decoupled, Space-decoupled-topic

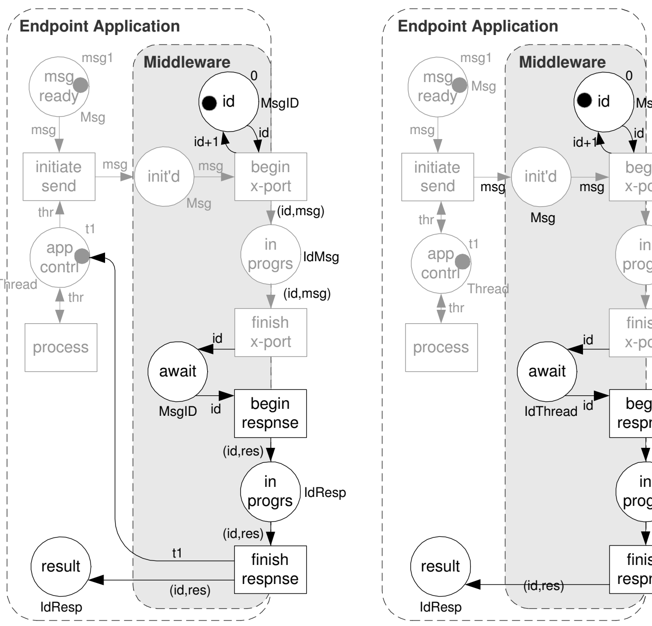
Fig. 15. Notations for the coupling integration patterns – space-decoupled topic.

The analysis of (de-) coupling has thus far only accounted for uni-directional interactions. Bi-directional interactions, the exchanging of data in two directions, occurs within the scope of a single interaction for RPC style middleware. We challenge the notion that only thread-coupled systems allow bi-directional messaging within an interaction.

Bi-directional messaging, adds additional possibilities – for instance delivery receipt, and the reporting of receiver-side faults. A delivery receipt (i.e. system acknowledgement) is an event returned to the requestor, that its message has been successfully received. A delivery receipt (as distinguished from a response) does not imply that the targeted endpoint has processed the message – just that it has received it. Finally a receiver side fault being propagated back to the requestor indicates that an error occurred during the processing of the request message. For simplicity, the CPNs presented here do not deal with networking faults, only with application-level faults. Again, for purposes of expedient presentation timeouts are not included in the following CPNs. Nevertheless we have built timeouts into our experimental CPNs.[6] To add timeouts to Figure 16(a) for example, we could make the CPN into a timed CPN. To do this we could make the following colour sets timed: `Msg, MsgID, IdMsg` and `IdResp`. This would allow us to simulate the passage of time while tokens are passed through

[6] Note also that the prototype mentioned in Section 7 does address timeouts and networking faults.

the model. We could add a sub-net "*timeout*", connected to the places of either the requester, or the responder. Should the interaction be incomplete when the timeout expires, the interaction is aborted, the places are cleaned up and a token is returned to the place "`app contrl`" – restoring control to the endpoint.



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

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

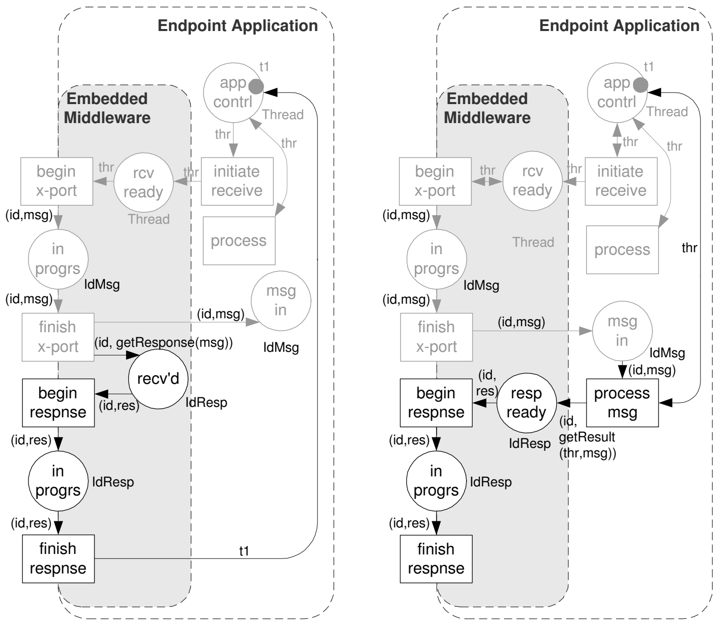
Fig. 16. Extended CPNs dealing with thread-coupling in bi-directional interactions from the requestor side. The changes made to the related CPNs from Section 3 are black, while the unchanged elements are grey.

Therefore users of time-decoupled solutions are typically forced to use work-arounds 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 threading 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 orthogonality of time, space, and threading. 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 Figure 16(b) continues processing immediately. The application in Figure 16(b) can rendezvous with the result when it is ready.

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 thread-coupling 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



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

Fig. 17. Extended CPNs dealing with thread-coupling in bi-directional interactions from the respondent side. The changes made to the related CPNs from Section 3 are black, while the unchanged elements are grey.

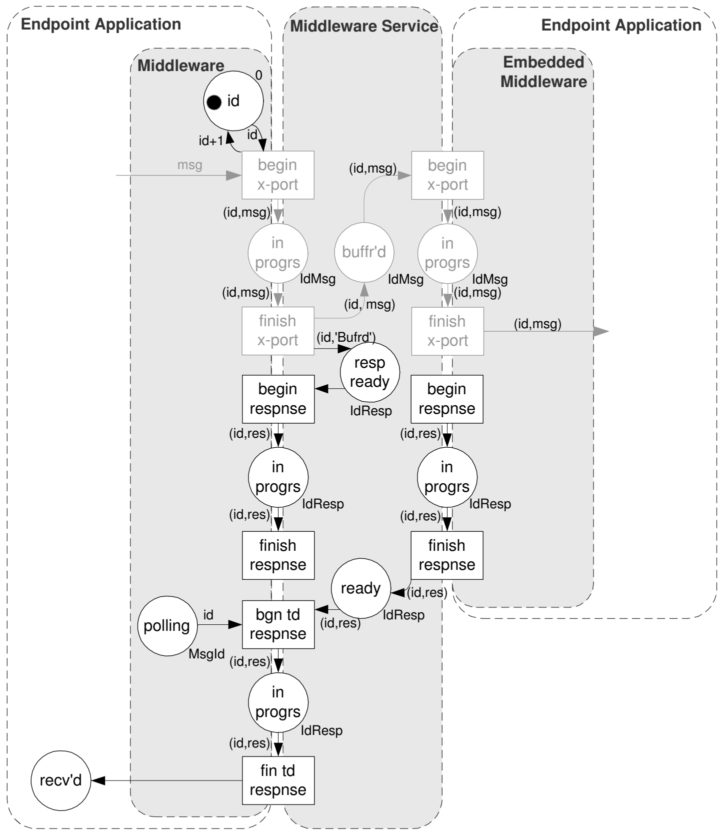


Fig. 18. Time decoupling CPN, extended to cover request-response interactions.

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 requestor indicating the message has been buffered, and is ready for the receiver to retrieve. If the case arises that the requestor still wishes to retrieve a response from the respondent 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 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 requestor indicating the successful receipt of the message.
- Postprocess acknowledgement – a signal, provided by the middleware, gets returned to the requestor indicating that the message was successfully processed.
- Postprocess response – a response, containing information provided by the respondent, gets returned to the requestor.
- Postprocess fault – an exception/fault occurs in the respondent while processing the message, and information about this gets propagated back to the requestor.
- Receive response: blocking – the requestor blocks for the response.
- Receive response: non-blocking – the requestor thread uses a non blocking technique to receive the response.

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 concepts presented in this paper, are supported by a range of middleware solutions and standards, but to varying degrees and in different combinations. This can make the selection of a middleware solution quite difficult, especially considering that the language used in different middleware "camps" can be difficult to penetrate, due to subtle changes in meaning of common words.

Therefore, in this section we propose that these concepts can be used to evaluate various middleware solutions and standards. Such an evaluation may be of assistance to architects deciding between technologies based on the requirements with respect to levels of coupling or decoupling between distributed systems. To illustrate this proposition, we have evaluated the following: Java-spaces[7], Axis [4], CORBA [13], JMS[8], Websphere MQ[9], MSMQ [20], and MPI[10].

First of all it is important to establish that with any of these standards and solutions, it is possible to implement all of the concepts. The question we are addressing is the relative level of effort required to achieve it. The results of this evaluation are presented in Table 1. Solutions that directly support a coupling integration 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.

In Table 1 rows 1 to 9 in the table cover the coupling integration patterns presented in figures 13, 14, and 15. The remaining rows correspond to the two patterns related to multicast (see Section 3.3) and six patterns related to the generation and handling of responses (see Section 5).

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.

7 Prototype

In the spirit of validating the proposal, we designed an API on top of the Java language, namely JCoupling [3], that embodies the coupling dimensions and integration patterns presented in earlier sections. Moreover, we developed a prototype implementation of this API and tested it on a number of scenarios. This prototype implementation is not a middleware per se: It does not provide application services traditionally associated with middleware such as reliable delivery (i.e. retries), transactions, security, etc. Its purpose is merely to illustrate how the proposed concepts can be used to support different types of communication through a unified API. The source code of the prototype is available from `www.sourceforge.net/projects/jcoupling`.

A JCoupling endpoint may change it's coupling with remote endpoints differently for each interaction, if the programmer chooses this. Thus it is possible, but not necessary, to send different messages using different coupling styles. JCoupling is written in Java 5, and makes extensive use of its new language features, including 'generics', and the new 'concurrent' tool-set.

Figure 19 presents a summary of the JCoupling API. It can be seen from this figure that the abstract class `Integrator` is central to the JCoupling API. Concrete implementations of `Integrator` perform the transport responsibilities, required by any `Sender` and `Receiver`. For instance, a possible implementation of `Integrator` could enable interactions over TCP sockets. The `JCouplingFactory` interface provides a dynamic means of creating instances of alternative implementations of the `Integrator` class. For instance an implementation of this interface could create either a JMS, TCP, or SOAP/HTTP `Integrator` implementation based on the contents of a text-based configuration file.

As shown in Figure 19, the `Integrator` interface has two primary methods:

– `receiveRequest()` immediately returns a `RequestKey` identifying the interaction request, and then places the request onto the JCoupling server (which is not shown). When a

[7] Java Spaces `http://java.sun.com/developer/products/jini/index.jsp`, accessed June 2006.
[8] J2EE-SDK V. 1.4, `http://java.sun.com/products/jms/`, accessed June 2006.
[9] Websphere MQ V 5.1, [19].
[10] MPI Core: V. 2, [24].

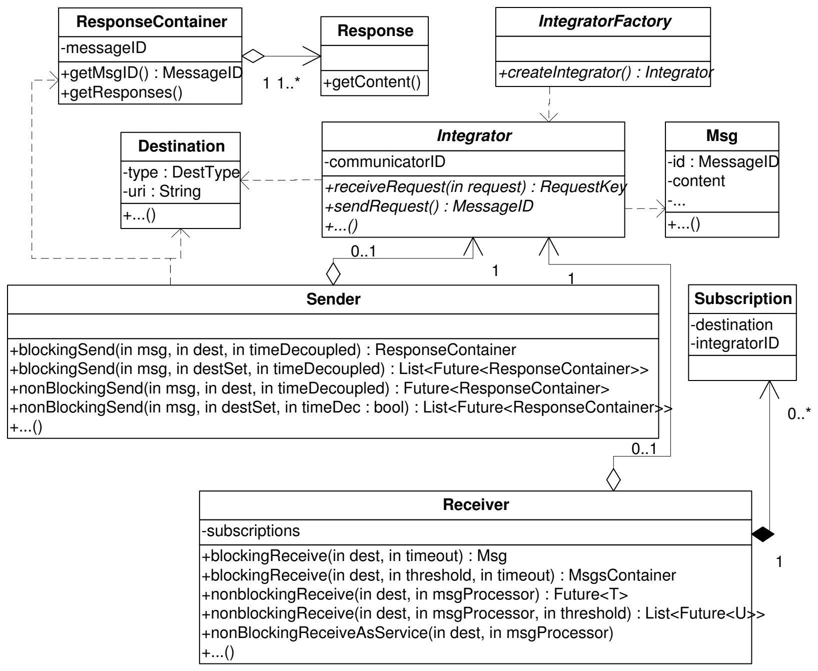

ResponseContainer
-messageID
+getMsgID() : MessageID
+getResponses()
1
1..*
Response
+getContent()
IntegratorFactory
+createIntegrator() : Integrator
Destination
-type : DestType
-uri : String
+...()
Integrator
-communicatorID
+receiveRequest(in request) : RequestKey
+sendRequest() : MessageID
+...()
Msg
-id : MessageID
-content
-...
+...()
0..1
1
1
Sender
+blockingSend(in msg, in dest, in timeDecoupled) : ResponseContainer
+blockingSend(in msg, in destSet, in timeDecoupled) : List<Future<ResponseContainer>>
+nonBlockingSend(in msg, in dest, in timeDecoupled) : Future<ResponseContainer>
+nonBlockingSend(in msg, in destSet, in timeDec : bool) : List<Future<ResponseContainer>>
+...()
Subscription
-destination
-integratorID
0..*
0..1
Receiver
-subscriptions
+blockingReceive(in dest, in timeout) : Msg
+blockingReceive(in dest, in threshold, in timeout) : MsgsContainer
+nonblockingReceive(in dest, in msgProcessor) : Future<T>
+nonblockingReceive(in dest, in msgProcessor, in threshold) : List<Future<U>>
+nonBlockingReceiveAsService(in dest, in msgProcessor)
+...()
1

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

message is ready, on the JCoupling server, a call-back is made by the server onto the requesting `Integrator` correlating the original request using the same key. This notifies the `Receiver` to retrieve the message off the server. The blocking or non-blocking interaction styles are implemented within the Receiver, which is akin to the way they were modelled in the CPNs of previous sections.

– `sendRequest()` operates in a similar manner, except that a `MessageID` is used instead of a request key, and that the client is a `Sender`.

Non blocking methods on both the `Sender` and the `Receiver` immediately return `java.util.concurrent.Future` objects. An object of type Future is essentially a handle to obtain a desired object once it is ready. The methods of `Sender` and `Receiver` that return lists are concerned with multicast and aggregate message receival.

Example 1: Hello World. 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.

```
...
IntegratorFactory factory = new LocalIntegratorFactory();
try{
    Integrator integrator = factory.createIntegrator();
    Sender sender = new Sender(integrator);
    Channel channel = (Channel) integrator.lookup(CHANNEL_URI);

    Message message = new Message();
    message.setContent("Hello World");

    sender.blockingSend(message, channel, TimeCoupling.decoupled);
} catch (JCouplingException e) { ... }
```

In Listing 1, line 2 creates a factory capable of instantiating `Integrator` objects, which basically allows client code to abstract away from the underlying transport protocol. 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 JCouplingException).

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

```
1     ...
2     try{
3         Integrator integrator = factory.createIntegrator();
4         Receiver receiver = new Receiver(integrator);
5         Channel channel = (Channel) integrator.lookup(CHANNEL_URI);
6
7         Message message = receiver.blockingReceive(channel, Receiver.NEVER_TIMEOUT);
8
9         // At this point, the message has been received
10        ...
11    }
12    catch (JCouplingException e) { ... }
13    catch (TimeoutException e) { ... }
```

In Listing 2, lines 3 - 5 create the receiver, and obtain a reference to the 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`

Example 2: "Scatter-gather". A "scatter-gather" interaction is akin to an RPC Broadcast: given a collection of endpoints, a request is sent to each of these endpoints, and a response is later gathered from each of them. For demonstration purposes we will portray a purchase order scenario. When a purchase order is received by *Hardware-R-Us* an inventory check reveals that certain line items are understocked. So *Hardware-R-Us* performs a scatter-gather request on three wholesalers for a quote, with the best quote being pursued.

Hohpe and Woolf [15] propose an implementation of the *scatter gather* pattern using JMS. The endpoint playing the role of *Hardware-R-Us* sends a request onto a JMS topic, and each wholesaler receives the request, creates a quote, parses the request for a return address, and posts the quote on the queue. The "*Hardware-R-Us*" endpoint receives each quote, one by one, keeping the best.

To implement the same scatter-gather using JCoupling requires less coding. *Hardware-R-Us* publishes a non-blocking-send, as shown in Listing 3.

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

```
1     ...
2     try {
3         IntegratorFactory factory = new LocalIntegratorFactory();
4         Integrator integrator = factory.createIntegrator();
5         Sender sender = new Sender(integrator);
6         Topic topic = (Topic) integrator.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
```

```
        ...// do something else while responses are coming back

        ResponseContainer responseContainer = futureResponses.get();
        List<Response> subscriberResponses = responseContainer.getResponses();
        List<Response> goodResponses = filterResponses(subscriberResponses);
        ...
    } catch (JCouplingException e) { ... }
    catch (ExecutionException e) { ... }
    catch (InterruptedException e) { ... }
```

The wholesaler endpoints don't need to explicitly receive messages, inspect return addresses, and send replies, they only need to implement an interface and add it to a `Receiver` object (Listings 4 and 5).

Listing. 4. The message processor interface.

```
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 to format a response. Method `processMessage` gets called first, then `getResponse`. The result of invoking `getResponse` gets returned to the sender.

Listing. 5. Creates a search receive/response server for a wholesaler.

```
1    try {
2        IntegratorFactory factory = new LocalIntegratorFactory();
3        Integrator integrator = factory.createIntegrator();
4        Receiver receiver = new Receiver(integrator);
5        Topic topic = (Topic) integrator.lookup(TOPIC_URI);
6        receiver.subscribe(topic, TOPIC_PASSWORD);
7
8        QuoteRequestProcessor quoteRequestProcessor = new QuoteRequestProcessor();
9        receiver.nonBlockingReceiveAsService(topic, quoteRequestProcessor);
10   } catch (JCouplingException e) { ... }
```

In Listing 5, lines 4 - 6 create the receiver, obtain the topic reference, and subscribe the receiver to the topic. Lines 8 - 9 instantiate an instance of the `MessageProcessor` interface and register it with the receiver. When a message arrives, the method `processMessage` of `QuoteRequestProcessor` will be invoked (similar to `onMessage` in JMS).

In JCoupling 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, JCoupling can guarantee preservation of message sequence. Another feature of JCoupling is its ability to perform multicast, and multiple message joining.

In building this prototype we now know that it is not overly difficult to build a coupling platform that natively incorporates the best features of Message-Oriented Middleware and RPC-based middleware, and that an implementation of these proposals does not need to compromise on the strengths of either middleware family. Furthermore we have demonstrated that the API to support this wide range of interaction styles can be relatively simple and concise.

8 Related Work

Cypher and Leu [10] 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 [24]. This work does not consider space decoupling. Our research differs by combining thread-decoupling with the principles of time and space-decoupling (originating from Linda [12]). Furthermore, our work unified these dimensions to define coupling integration patterns that can be used as a basis for middleware comparison.

Charron-Bost, Mattern, and Tel [7] provide a formalisation of the notions 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 thread-coupled, time-coupled communication).

Cross and Schmidt [9] 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 [26] 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 [23] described four classes of middleware: *Host infrastructure middleware* (e.g. sockets), *Distribution middleware* (e.g. CORBA [13], and RMI [17]), *Common Middleware Services* (e.g. CORBA and EJB), and *Domain Specific Middleware Services* (e.g. EDI[11] and SWIFT[12]). 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 [25] 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. [5] produced a set of service interaction patterns. The paper is oriented towards Web services and their relationship with technologies for implementing conversational services such as the Business Execution Language for Web Services (BPEL4WS). However, these service interaction patterns do not deal with the notion of coupling. Also, the service interaction patterns focus on the issue of dealing with state management between interactions, as opposed to dealing with the characteristics of individual interactions.

[11] EDI: Electronic Data Interchange is a set of standards defining electronic document structure for business to business communication. There are two standards sets for EDI, one adopted by the ISO `http://www.unece.org/trade/untdid/welcome.htm`, and one adopted by ANSI `http://www.x12.org/` (accessed Jan 2007).

[12] SWIFT: Society for Worldwide Interbank Financial Telecommunication. SWIFT provide a value added network enabling messages concerning transactions to be exchanged between banks of the world (`http://www.swift.com` accessed Jan 2007).

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.

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: (1) threading, (2) time and (3) space. 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. [15]). However, such definition does not provide any differentiation between sockets and RPC, which are both time-coupled. In this paper, we have argued that the terms 'synchronous' and 'asynchronous' are too imprecise to constitute a foundation for defining models of integration. This had led us to define twenty-four coupling integration patterns that provide a more precise definition of the coupling aspect of integration models. They could also be used as a middleware selection instrument. This set of patterns unifies and organises existing knowledge in the domain of integration coupling.

We have also presented an embodiment of the proposed concepts in the form of an API, namely JCoupling. JCoupling is similar in many ways to JMS, but it extends it to cover the identified coupling dimensions and associated concepts.

In ongoing work, partly reported in [3], we have further extended the JCoupling API to deal with message filters, and we have shown how this enables the implementation of sophisticated forms of message correlation in the context of business process management systems. We are currently extending this work to integrate it into a workflow engine, namely YAWL [27].

Disclaimer The assessments we made of middleware products and standards with respect to the coupling integration patterns are based on the tool or standard's documentation and, in some cases, experimentation. 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 *Proceedings of the 7th International Symposium on Distributed Objects and Applications (DOA)*, pages 1015 – 1033. Springer Verlag, November 2005.
3. L. Aldred, W. van der Aalst, M. Dumas, and A. ter Hofstede. Communication Abstractions for Distributed Business Processes. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering.*, volume 4495 of *Lecture Notes in Computer Science*, pages 409–423. Springer Verlag, 2007.
4. Apache axis homepage. `http://ws.apache.org/axis/` accessed Sept 2007.
5. A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns. In *Proceedings of the 3rd International Conference on Business Process Management (BPM)*, pages 302–318. Springer Verlag, September 2005.
6. A. Beugnard, L. Fiege, R. Filman, E. Jul, and S. Sadou. Communication Abstractions for Distributed Systems. In *ECOOP 2003 Workshop Reader*, volume 3013, pages 17 – 29. Springer-Verlag, Berlin, 2004.

7. B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996.
8. CPN Tools homepage. `http://wiki.daimi.au.dk/cpntools/` accessed Sept 2007.
9. J. K. Cross and D. 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.
10. 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.
11. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
12. D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
13. Object Management Group. *Common Object Request Broker Architecture: Core Specification*, 3.0.3 edition, March 2004. `http://www.omg.org/docs/formal/04-03-01.pdf` accessed Sep 2007.
14. M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Haase. *Java Messaging Service API Tutorial and Reference.* Addison-Wesley, 2002.
15. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, Boston, MA, USA, 2003.
16. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1.* EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
17. Sun Microsystems. Java remote method invocation specification. `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html` accessed Sept 2007.
18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised.* MIT Press, Cambridge, USA, 1997.
19. Websphere MQ family. `http://www-306.ibm.com/software/integration/wmq/` accessed Sept 2007.
20. Microsoft Message Queuing. `http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx` accessed Sept 2007.
21. 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.
22. E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
23. 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.
24. M. Snir, S. Otto, D. Walker S. Huss-Lederman, and J. Dongarra. *MPI-The Complete Reference: The MPI Core.* MIT Press, second edition, 1998.
25. A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
26. J. Thompson. Toolbox: Avoiding a middleware muddle. *IEEE Software*, 14(6):92–98, 1997.
27. YAWL Home Page. `http://www.yawlFoundation.com` accessed September 2007.

Dimension	**Option**	#	Java Spaces	Axis	CORBA	JMS	Websphere MQ	MSMQ	MPI
Threading	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	+/−	−	+/−	+	+	+/−	+/−
Multicast									
Multicast/Scatter		M1	−	−	−	−	+	+	+
JoinMsgs/Gather		M2	−	−	−	−	−	−	+
Request-Response									
PreprocessAck		R1	−	−	−	+	+	−	+
PostProcessAck		R2	−	+	+	+/−	+/−	−	−
PostProcessResp		R3	−	+	+	+/−	+/−	−	−
PostProcessFault		R4	−	+	+	−	−	−	−
Blocking Receive Ack		R5	−	+	+	+	+	−	−
Non Blocking Receive Ack		R6	−	−	+	−	−	−	−

Table 1. Evaluation of selected middleware solutions and standards.

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 coupling capabilities presented in this article. 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, are non-trivial.

A.1 Java Spaces

Java Spaces supports a blocking-send through its `write` operation. However it does not support a non-blocking send, hence capability '2' was given a '–' in Table 1.

Java Spaces supports blocking-receive through its `read` or `take` operations, and supports non-blocking receive through its `notify` operation, hence capabilities '3' and '4' were assigned a '+' in Table 1.

Java Spaces, is time-decoupled, and is not time-coupled. Consequently capability '5' was assigned a '–' in Table 1.

Java Spaces supports space-decoupling over a channel, but does not support coupling one sender to one receiver, hence a '–' for capability '7' in Table 1.

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 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 a 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. Consequently non-blocking send and blocking receive were assigned a '–' 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, items '8' and '9' were assigned '–'.

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

Axis supports many form of acknowledgement, hence it was well represented in the list of items of Table 1 related to responses. Nevertheless, 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 1).

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

CORBA, like Axis would require thread forking techniques to provide the requestor with an acknowledgement before the request gets processed, therefore we gave response pattern R1 a '–' in Table 1. Otherwise CORBA has the best representation of any of the solutions or standards evaluated for supporting the various forms of response, and that is reflected in Table 1.

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, and middleware service on the same machine. The message is passed into the middleware service and then is put over the network without the sender needing to wait. Hence we assign JMS a '+/–' for item '2' in Table 1.

JMS, being a MOM driven standard, natively supports time-decoupling but not time-coupling. Hence a '–' for item '5'.

JMS supports both forms of space-decoupling (channel and topic), but not space-coupling. Consequently item '7' in Table 1 was 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).

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 item ‘7’ in Table 1 was assigned a ‘+’.

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 item ‘2’ in Table 1.

MSMQ, like most MOM solutions does not support time-coupling. Hence we assigned a ‘–’ to items ‘5’ and ‘6’ in Table 1.

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 item ‘7’ in Table 1.

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 (or publish-subscribe) is a work-around, in much the same class as Java spaces. Therefore, it was assigned a ‘+/–’ for item ‘9’ in Table 1.

MSMQ, supports multicast directly but to our knowledge does not support multicast-gather, or any of the 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 thread (de-) coupling, providing explicit operations for blocking-send, non-blocking send, blocking-receive, and non-blocking receive – as is shown in Table 1.

MPI does not support time-decoupling – as shown in Table 1.

MPI does not support space-decoupling over a channel.

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.