On the Notion of Coupling in Communication 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 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 the application components are generally tightly coupled between them and with the underlying communication 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 along different dimensions. In the literature, terms such as synchronous, asynchronous, blocking, non-blocking, directed, and non-directed are generally used to refer to the degree of coupling required by a given architecture or provided by a given middleware. However, these terms are used with various connotations by different authors and middleware vendors. And while several informal definitions of these terms have been provided, there is a lack of an overarching framework with a formal grounding upon which software architects can rely to unambiguously communicate architectural requirements with respect to coupling. This paper addresses this gap by: (i) identifying and formally defining three dimensions of coupling; (ii) relating these dimensions to existing communication middleware; and (iii) proposing notational elements for representing coupling configurations. The identified dimensions provide the basis for a classification of middleware which can be used as a selection instrument.

1 Introduction

Distributed application integration has some longstanding problems. For instance technology and standardization efforts supporting distributed interactions (Web services [3], MOM [11], MPI [20], RPC/RMI [14]) have made it possible to implement solutions to the difficult problems of distributed communication, however a general framework/theory for integration remains elusive. The problem seems to be one of finding the *right* abstractions [2].

Researchers seems to agree that the problem of communicating autonomous systems is not well understood [6, 2, 7, 13] and yet middleware vendors appear

confident that the problem is well understood, at least with respect to their tools. While there are incredibly complex problems yet to be solved in the area of middleware, perhaps those issues and aspects related to coupling/decoupling lie at the very heart of the problem. In their paper about the semantics of blocking and non-blocking send and receive primitives, Cypher & Leu state that "unfortunately, the interactions between the different properties of the send and receive primitives can be extremely complex, and as a result, the precise semantics of these primitives are not well understood." [6].

Vendors and standards bodies offer many solutions and ideas within this domain, however each appears to be embroiled in the paradigm it emerged from. For instance CORBA is founded and based on an RPC paradigm, whereas MOM [1] platforms are based on a strongly decoupled paradigm involving a hub architecture, with a MOM service at the centre. Despite their different origins each appears to have technical and conceptual limitations.

Due to the lack of a widely accepted formal theory for communication and the fact that each type of solution is based on a totally different paradigm, models and implementations of middleware based communication are disparate, and not portable. Rephrased, the models over deployed distributed systems are not portable in the implementation sense, and not even in the conceptual sense. Due the lack of formality in this domain, solutions and tools for integration are difficult to compare as well.

Objectives of the Paper
This paper aims at contributing to address the lack of foundation for expressing architectural requirements and for assessing middleware support in terms of decoupling. The main contributions of the paper are:

- A detailed analysis of the notion of (de-)coupling in communication middleware.
- A collection of notational elements for expressing architectural requirements in terms of (de-)coupling. These notational elements are given a visual syntax extending that of Message Sequence Charts [18] and can thus be integrated into UML sequence diagrams.[2]. In addition, the notational elements are given a formal semantics in terms of Coloured Petri Nets (CPNs) [12].
- An approach to classify existing middleware in terms of their support for various forms of (de-)coupling. This classification can be used as an instrument for middleware selection.

Scope
A complete, formal analysis of communication middleware would be a daunting task. The list of options and functionality of middleware is incredibly long, particularly when one considers, for example privacy, non-repudiation, transactions,

[1] Message Oriented Middleware: middleware systems capable of providing support for decoupled interactions between distributed endpoints.
[2] http://www.uml.org

reliability, and message sequence preservation. Therefore this work chooses not to take too broad an analytical view of the domain. We have chosen to focus in on the aspect of *decoupling* because it seems to lie at the very heart of the problem of making two or more endpoints communicate effectively, and is central to the design of distributed applications.

In a recent survey of publish-subscribe technologies Eugster [7] identified three primary dimensions of decoupling offered by MOM. These are:

- *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 address of a message is directed to a particular symbolic address (channel) and not the direct address of an endpoint.
- *Synchronisation Decoupling* - wherein the threads inside an endpoint do not have to block (wait) for an external entity to reach an appropriate state before message exchange may begin.

These three dimensions of decoupling, we believe, exist at the very core of middleware functionality. Despite their crucial role they are not well understood [6], and to the best of our knowledge have not been the subject of a formal analysis.

Organisation of the Paper
The paper is structured as follows. Section 2 defines some basic concepts and reviews related work. Section 3 identifies a set of decoupling dimensions and defines basic elements for expressing architectural requirements in terms of these dimensions. Next, Section 4 shows how these elements, as well as their formal definitions in terms of CPNs, can be composed in order to capture requirements across multiple coupling dimensions. Finally, Section 5 concludes and outlines directions for further work.

2 Background

This section provides an overview of background knowledge related to the study of decoupling in communicating middleware. It first defines some essential terms used throughout the paper, and proceeds with an overview of related work.

2.1 Definitions

An *endpoint* is an entity that is able to participate in interactions. It may have the sole capability of sending/receiving messages and defer processing the message to another entity, or it may be able to perform both.

An *interaction* is an action through which two endpoints exchange information [17]. The most basic form of this occurs during a message exchange (elementary interaction).

A *channel* is an abstraction of a message destination. Middleware solutions such as JMS [10], WebsphereMQ [15], and MSMQ [16] use the term "queues" to mean basically the same thing, but the crucial point here is that a channel is a logical address, not a physical one, thus they introduce a space decoupling to traditional point-to-point messaging. Thus the sender is able to address a message to a symbolic destination, and the receiver may register the intention to listen for messages from the symbolic destination. Interestingly such a decoupling means that a channel is not necessarily restricted to one message receiver - many *endpoints*, may share one channel, in which case they may either share/compete for each message, depending on whether or not the channel is a publish-subscribe.

Channels have been enhanced and extended with many functions, including the preservation of message sequence [7, 6], authentication and non-repudiation [11].

A *message* is a block of data that gets transported between communicating endpoints. Depending on the middleware it could contain header elements such as an message ID, timestamp, and datatype definition; or it could just contain data. The message may contain a command, a snapshot of state, a request, or an event among other things. It may be transactional, reliable, realtime, or delayed, and it often is transported over a "channel". However, it could just as easily be sent over sockets.

2.2 Related Work in Middleware Classification

Cross and Schmidt [5] discussed a pattern for standardizing quality of service control for long-lived, distributed real-time and embedded applications. This proposal briefly described a technology that would be of assistance. They outlined the technology as "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 paper perhaps provide a fundamental basis for the selection of compatible sets of infrastructure.

Schantz and Schmidt [19] described four classes of middleware: *Host infrastructure middleware* provides a consistent abstraction of an operating system's communication and concurrency mechanisms (e.g. sockets). *Distribution middleware* abstracts the distribution, and generally allows communication over heterogenous systems as if they were running in one stand-alone application (e.g. CORBA [9], and RMI [14]). *Common Middleware Services* group together middleware that can provide higher level services such as transactions, and security (e.g. CORBA and EJB). *Domain Specific Middleware Services* classify those that are tailored to the requirements of a specific real world domain, such as telecom, finance etc. (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.

Thompson [21] 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, ranging from conversational, through request-reply, to messaging, and finally to 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, as discussed later in the paper.

Cypher and Leu [6] provided a formal semantics of blocking/non-blocking send/receive which is strongly related to our work. These primitives were defined in a formal manner explicitly connected with the primitives used in MPI [20]. This work does not deal with space decoupling. Our research adopts concepts from this work and applies them to state of the art middleware systems. Our research differs from the above, by exploiting the knowledge of this work in terms of non-blocking interactions, and combining this with the principles of time and space decoupling originating from Linda [8]. Our work is also unique in its proposal for compositional dimensions of decoupling, and our communication primitives may be used as a basis for middleware comparison.

3 Decoupling Dimensions of an Interaction

In this section we will provide a Petri net-based analysis of fundamental types of decoupling for interacting endpoints. These dimensions of decoupling have relevance to all communication middleware we are familiar with, including MOM, space-based middleware [8], and RPC-based paradigms.

3.1 Synchronisation

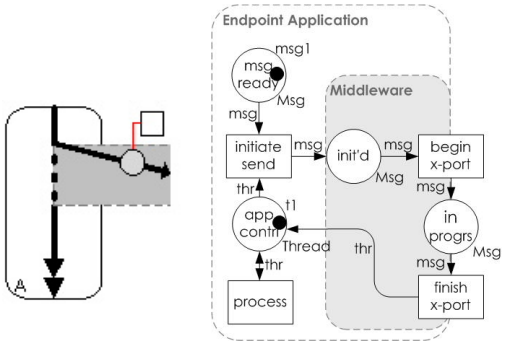
The critical concept behind synchronisation decoupling is that of "non-blocking communication", for either, or both of, the sender and receiver. Non-blocking communication allows the endpoints to easily interleave processing with communication. In the following paragraphs we introduce some notational elements for denoting various forms of synchronisation decoupling as well as a formalisation of these notational elements in terms of CPNs.

Send
A message send can either be blocking or non-blocking. *Blocking send* implies that the sending application must yield a thread until the message has truly left it. Figure 1(b) is a coloured Petri net [3] 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. 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

[3] Note: This paper presents a range of coloured Petri nets (CPNs) [12] that model important aspects of decoupled systems. All CPNs were fully implemented and tested using CPN Tools [4].

embedded middleware. The endpoint in blocking send also yields its thread of control to the embedded middleware, but it gets the thread back when the message has completely left the embedded middleware. Note that inside the embedded middleware the transitions "*begin-x-port*", "*in-progress*", and "*fin-x-port*" hang over the edge of the embedded middleware. This was done to demonstrate that the remote system (receiver endpoint or middleware service) will bind to the sender by sharing these transitions and one state. This implies that hidden inside the middleware, communicating systems exchange information in a time coupled, synchronisation coupled manner, regardless of the behaviour that's exposed to the the endpoint applications. In CPN terminology certain nodes inside the embedded middleware are "transition bounded" [4].

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.



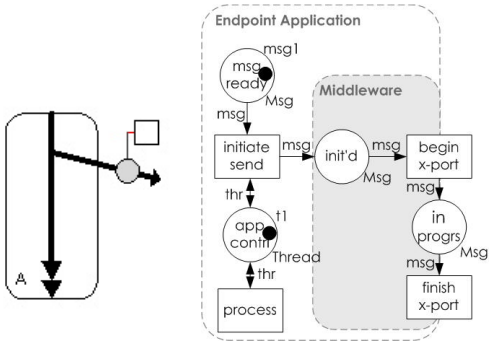
(a) Notation. (b) Petri net - Blocking send.

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

Synchronisation decoupling is achieved from the viewpoint of the sender only if it is possible to perform a *non-blocking send*. A non-blocking send is observable in a messaging interface if the message send operation can be initiated from the application and then the middleware embedded inside the sender returns control immediately, (i.e. before the message has left the application place). See Figure 2 for an illustration and Petri net model of the concept. Note that 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 [20].

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-

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



(a) Notation. (b) Petri net - 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.

blocking) then a total synchronisation coupling (decoupling) occurs. A partial synchronisation decoupling occurs when the send and the receive are not of the same blocking mode (i.e. one is blocking with the other being non-blocking).

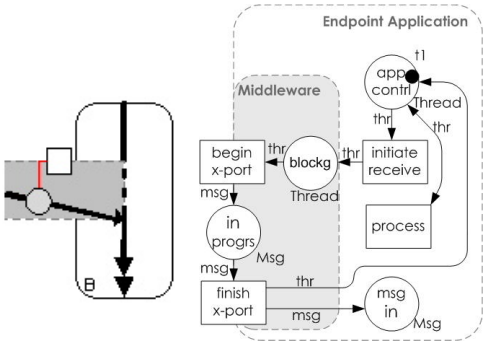
Non-blocking send is a fairly uncommon feature of middleware solutions. For instance all RPC-based implementations use blocking send, and many/most MOM implementations use a blocking send as well. This is the case because even though MOM decouples the sender from the receiver through time, the senders are typically synchronisation coupled to the middleware service. This is acceptable when the sender is permanently connected over a reliable network to the provider, however mobile devices, for instance, typically need to inter-operate on low availability networks, hence they require the ability to store the message locally, until the network is available (i.e. store and forward) [13]. This problem should obviously not be too great a burden on the mobile applications developer, and should be part of the middleware functionality.

Receive

Like message send, message receipt can either be blocking or non-blocking [6]. The definition of *blocking receive* is that the application must yield a thread into a waiting state in order to receive the message (the thread is usually returned 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 the whether the middleware is peer-to-peer or server-client oriented). Figure 3 presents a model of this concept.

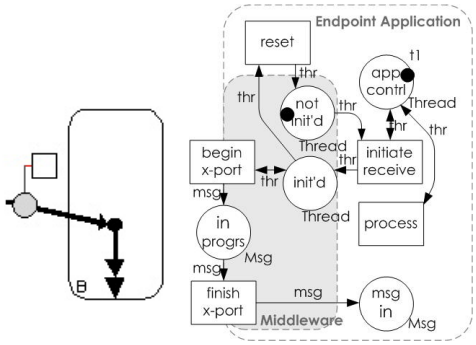
On the other hand, *non-blocking receive*, is a messaging concept wherein the application can receive a message, without being required to yield a thread to achieve this. This concept is illustrated in Figure 2.

A well known example of non-blocking-receive is that of the event-based handler, as described in the JMS [10]. A handler is registered with the middleware



(a) Notation. (b) Petri net - Blocking receive.

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



(a) Notation. (b) Petri net - non-blocking receive.

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

and is called-back when a message arrives. MPI provide an equally valid non blocking receive that is not event based [20].

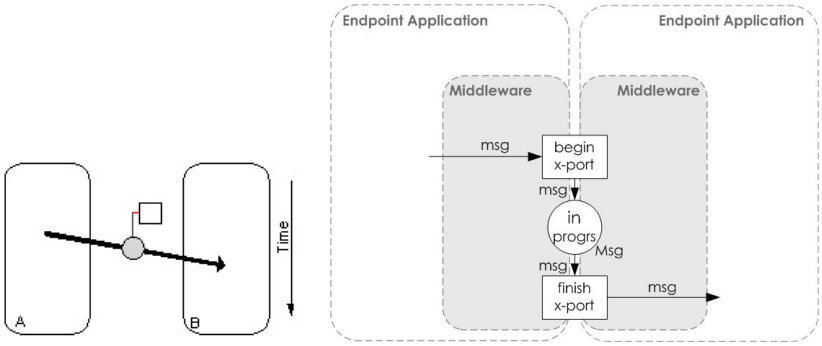
Non-blocking receive seems less frequently used than the blocking receive. This is probably because blocking receives are simpler to program and debug [11]. One frequently observes statements in the developer community that MOM enables asynchronous interactions (which is true in the it allows time decoupled interaction), however this general usage of "asynchronous" for MOM is misleading because MOM usually connects endpoints with a blocking-send and blocking receive (synchronisation coupled).

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 interactions are observable when communication cannot take place unless both endpoints are operating at the same time. Hence peer-to-peer systems are always time coupled. 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*").



(a) Notation. (b) Coloured Petri net of time coupled messaging.

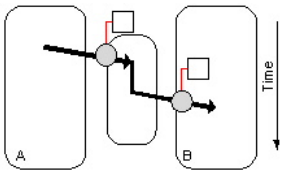
Fig. 5. *Time coupling* is characterised by transition-bounded systems.

Time decoupled interactions allow messages to be exchanged irrespective of whether or not each endpoint is operating at the same time. Therefore simple peer-to-peer architectures cannot provide true time decoupling - by definition. What is required for time decoupling is a third participant in the interaction where the sender can deposit the message, and the receiver can retrieve it. This is why many MOM implementations use a client-server architecture. Servers may be redundant, and even use "store and forward" semantics between servers [15], in which case the term "peer-to-peer" is often used ambiguously. Though this could be better described as a polygamous architecture where many machines host both peer-endpoints, and a messaging server.

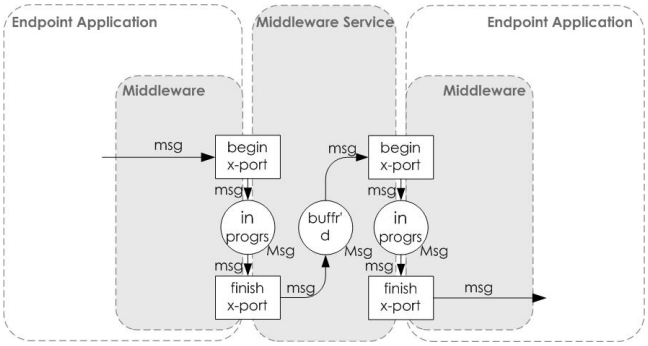
The concept of time decoupling is presented in Figure 6. Note that in the Petri net and the illustration the separate systems are transition bounded in the same way as before, however this time there are three of them, with the middle one being a middleware service that is able to buffer the message.

3.3 Space

Space coupling is the final dimension of decoupling considered to play a role in this domain.



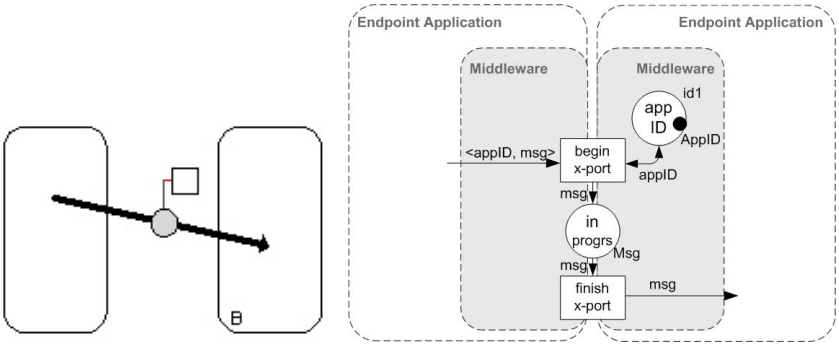
(a) Notation.



(b) Coloured Petri net of time decoupled messaging.

Fig. 6. *Time decoupling* is characterised by systems that can be strictly non-concurrent (endpoint to endpoint), and still communicate.

For an interaction to be *space coupled* the sender uses a direct address to send the message to. Therefore the sender "knows" exactly where to address the receiver application. Figure 7 presents the concept of space coupling. This can be seen in the Petri net by introducing a new type of token ($< appID, msg >$), and a new place ("*id*") as input to the transition "*begin x-port*". Only when two conjoined systems match on the value of "*appID*" will the bounding transition fire.



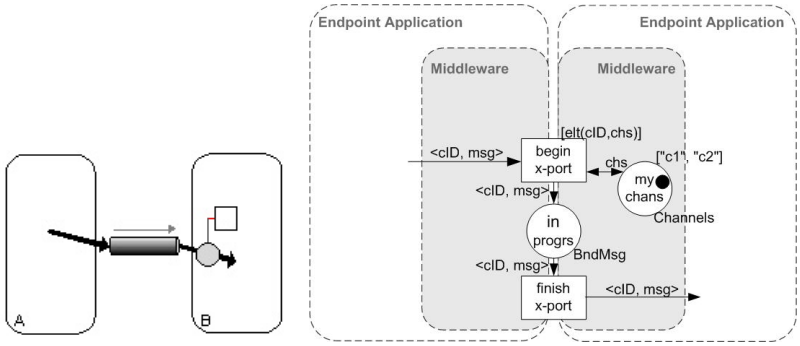
(a) Notation

(b) Coloured Petri net of space coupled messaging.

Fig. 7. *Space coupling.* The sender directly addresses the receiver.

Space decoupled interactions on the other hand allow for a sender to have no explicit knowledge of the receiver's address. This makes it possible for parts of distributed systems to be extended, and replaced at runtime without shutting everything down. Hence space decoupling is highly desirable from the viewpoint of enterprise integration due to its support for maintenance and management.

Figure 8 introduces the concept of an abstract message destination - or channel for space decoupled point to point messaging[5]. This is the logical message destination, but the actual message destination obtains its message off the same channel. The Petri net demonstrates this by linking the transition "*begin x-port*" to a new input place ("*my chans*") and a slightly different token containing the message ($< cID, msg >$), of type BoundMessage (BndMsg). Hence, this transition shall only fire when two systems are bound together that satisfy the transition guard "*[elt(cID,chs)]*".



(a) Notation. (b) Coloured Petri net of space decoupled messaging.

Fig. 8. *Space decoupling.* The sender does not directly address the receiver, but directs the message along a channel.

3.4 Summary

The dimensions of decoupling include synchronisation-decoupling (with its four options), time-decoupling, and space-decoupling. Each has its own precise behaviour and semantics. These were rendered using a reasonably intuitive graphical notation and a more precise formal semantics as presented by the Petri nets.

4 Combining Synchronisation, Time, and Space

The dimensions of decoupling presented in the previous section are orthogonal to each other. Therefore designs for interactions can be composed from them

[5] Note that this series of CPNs models point to point messaging (i.e. as opposed to publish-subscribe). Extending the models to describe publish-subscribe, while beyond the scope of this paper, is possible but not trivial.

arbitrarily while preserving the innate semantics, as fully defined for each - contributing to a precise overall behaviour. This set of configurations can then be used as a palette of possible interaction behaviours and thus applied to an integration problem or to the selection of an appropriate middleware product.

4.1 Compositional Semantics

Any type of synchronisation-decoupling (for both send and receive) can be combined with any type of time-decoupling, which in turn can be combined with any type of space-decoupling. This means that for one directional messaging there are sixteen possible interaction behaviours, definable according to the decoupling properties ($2^2 * 2 * 2 = 16$), and we believe that these dimensions are orthogonal. Meaning, for example, that you can have a time-coupled, synchronisation-decoupled interaction, and it is equally possible to have a time-decoupled synchronisation-coupled interaction.

Composing Petri nets
The Petri nets for each coupling dimension from Section 3 can also be composed, or *overlayed* to form a complete model of any of the sixteen possible interaction behaviours. For example to create a CPN of a synchronisation-decoupled, time-decoupled, and space-decoupled interaction one may use the CPNs from Figures 2(b), 4(b), 6(b), and 8(b), and overlay them. Such a net is presented in Figure 10. A CPN for a synchronisation-coupled, time-coupled, space-coupled also shown (Figure 9), however, for space reasons we do not present the remaining fourteen CPNs.

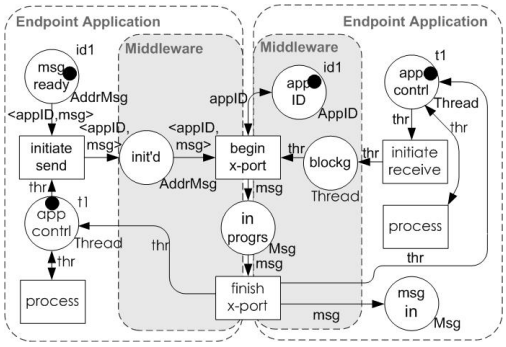


Fig. 9. Petri net of a synchronisation-coupled, time-coupled, space-coupled interaction between endpoints.

Graphical Notation of Compositions
The entire set of sixteen possible decoupling configurations possible are enumerated in graphical form in Figures 11 and 12. These graphical illustrations of the possibilities are essentially arrived at by overlaying the illustrative vignettes of the dimensions of decoupling as presented in Section 3.

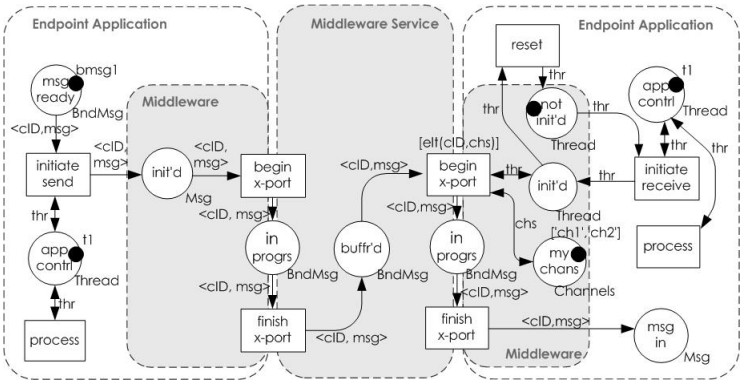


Fig. 10. Petri net of a synchronisation decoupled (non-blocking send/receive), time-decoupled, and space-decoupled interaction.

This graphical notation for the different types of coupling for elementary interactions could prove to be useful in defining requirements, or system analysis and design. The configurations are varied, and each one has its own specific behaviour. Furthermore they are sufficiently different that some will be more suitable to a given integration problem than other. Put another way not all possible coupling configurations would be useful in any situation. For instance a multi-player realtime strategy game would not have much use for a time-decoupled configuration.

4.2 Example of Capturing Coupling Requirements in an Integration Project

Imagine that a hospital needs to integrate a new BPM system and a new proximity sensor system to its existing IT services. Each doctor, and nurse is given a mobile device which can inform a central system of the location of that person inside the hospital. This is linked to the BPM system so that the nearest staff member with the requisite skills can be notified of new work and notified during emergencies.

The challenge is to design a conceptually clean, integration model showing the types of connectivity between the different endpoints in such a system. Clearly the mobile devices will not always be connected to the central systems (due to possible signal interference), and therefore non-blocking send is advisable, this way messages from the device could be stored until the signal is restored. New mobile devices might need to be added to the system, and device swapping may occur, and shouldn't break the system. Therefore space decoupling is required. Finally, device batteries might go flat and therefore time decoupling between mobile devices and the central system is necessary. Hence the architecture of this endpoint interconnection should be either configuration '14' (Non-blocking-send, Blocking-receive, Time-decoupled, Space-decoupled) or '16' (Non-blocking-send, Non-blocking-receive, Time-decoupled, Space-decoupled). During requirements

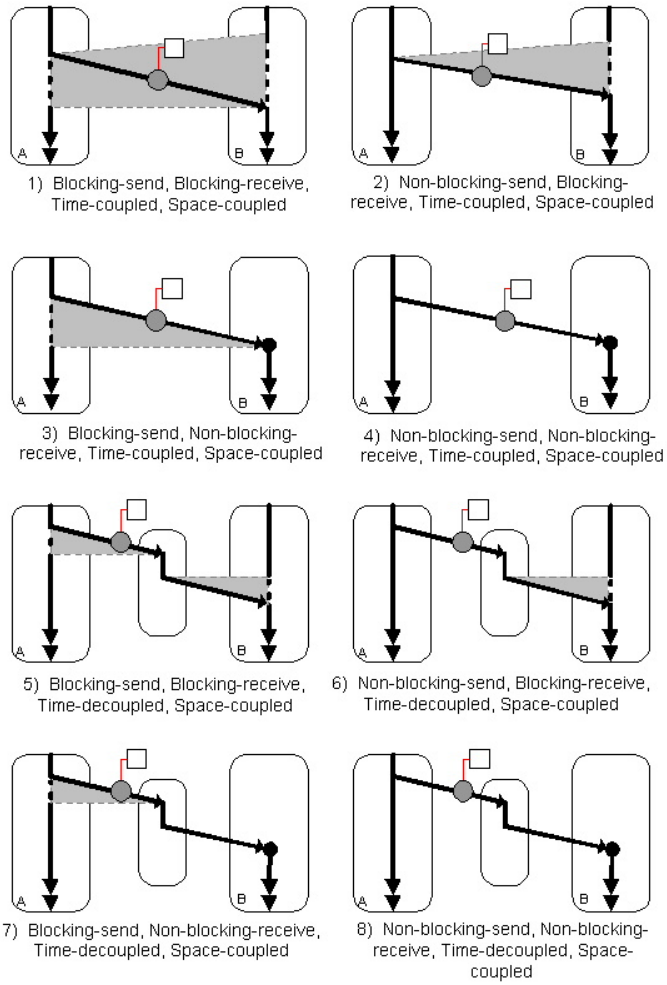


Fig. 11. Notations 1 - 8 of the coupling configurations for one way communication.

analysis we can proceed through each endpoint connection using a similar approach.

4.3 Comparison of Middleware Systems

We postulate that any type of middleware could be plotted against the 16 coupling configurations proposed in this Section with respect to whether they directly support the defined behaviour through their API or interface. As part of this research we have plotted the coverage of middleware solutions and standards against the proposed coupling configurations. The set of solutions and stan-

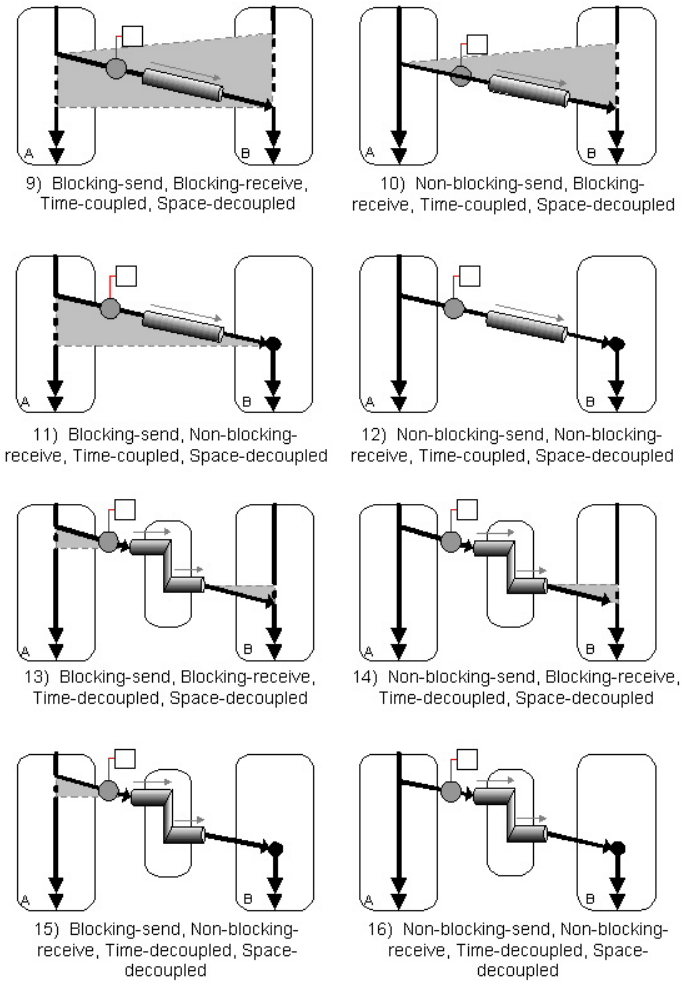


Fig. 12. Notations 9 - 16 of the coupling configurations for one way communication.

dards includes MPI[6], Biztalk Server 2004 (Product Documentation), Websphere MQ[7], Java-NIO[8], Java-RMI[8], Java-Sockets[8], Java-Mail[9], JMS[9], Java-spaces[10], CORBA [9], RPC[11]. In most cases the documentation (as opposed to implemen-

[6] MPI Core: V. 2, [20].
[7] Websphere MQ V 5.1, [15].
[8] JDK V. 1.4, `http://java.sun.com/j2se/1.4.2/docs/api`, accessed June 2005.
[9] J2EE-SDK V. 1.4, `http://java.sun.com/j2ee/1.4/docs/api`, accessed June 2005.
[10] Java Spaces `http://java.sun.com/products/jini`, accessed June 2005.
[11] DCE-RPC V 1.1, `http://www.opengroup.org/onlinepubs/9629399/toc.htm`, accessed June 2005.

tations) for these standards and solutions was used as a guide to determine their coverage.

Space coupling	Synch coupling	Partial synch decoupling		Synch decoupling
	B-Send, B-Rcv	NB-Snd, B-Rcv	B-Snd, NB-Rcv	NB-Snd, NB-Rcv
Time coupled	MPI, Sockets 1	MPI, RPC-Reply 2	MPI, CORBA, RPC-Request 3	MPI, Java-NIO 4
Time decoupled	Java-Mail 5	6	7	8

Space decoupling	Synch coupling	Partial synch decoupling		Synch decoupling
	B-Send, B-Rcv	NB-Snd, B-Rcv	B-Snd, NB-Rcv	NB-Snd, NB-Rcv
Time coupled	9	Java-RMI-Reply 10	CORBA, Java-RMI-Request 11	12
Time decoupled	JMS, Websphere-MQ, BizTalk-MSMQ, JavaSpaces 13	14	JMS, Websphere-MQ, BizTalk-MSMQ, JavaSpaces 15	16

Table 1. Support for coupling configurations by some well known middleware solutions and standards.

Table 1 represents our initial assessment of various well known middleware solutions and assesses each one's ability to directly support each coupling configuration (hence that communication behaviour).

The hospital scenario previously introduced, requires either configurations 14 or 16, these are both empty in our tables, however MobileJMS [13] is a proposal that will support them.

4.4 The Case of Two-way Interactions

This work, while presented in terms of one directional communication, has application in compound interactions as well - for instance two-way communication. In RPC style interactions there are two roles being played. The role of requestor performs a "solicit-response", and the role of service provider performs a "request-response" [1]. The requestor typically blocks for the response, hence the interaction is generally considered synchronous. However, if such an interaction is modelled using the proposed notation it becomes clear that this broad definition is not totally precise. In actual fact the interaction is partially synchronisation decoupled, in each direction. The service provider uses non-blocking receive and in the reply uses non-blocking send (B-send → NB-receive → NB-send → B-receive). Therefore any model of such an interaction should capture these subtleties.

Using the proposed notation there are 16 possibilities in each direction for two way interactions. Hence if one enumerated all possible configurations of decoupling, for two way interactions, there are $16^2 = 256$ possibilities.

5 Conclusions

This paper 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 communication 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 communication 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. [11]). However, such an imprecise definition does not provide any differentiation between sockets and RPC, which are both time-coupled. Clearly there is more to the problem than the generally held belief (that time-coupled implies synchronous). We consider that 'synchronous' and 'asynchronous' are too imprecise for using to create clear abstract models of integrations. The framework that we provide is the first that we know of that presents synchronisation coupling and time coupling as independent but related concepts.

Currently the rating of tools against the coupling configurations are binary, while perhaps a recognition of partial support (where the tool fully implements a restricted version of the concept, or it supports it, but requires some extra modelling effort to implement it) would make more useable. The assessment would also benefit from a more detailed explanation of why a rating was given.

Disclaimer The assessments we made of middleware products and standards with respect to the coupling configurations 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 Government Smart State Fellowship.

References

1. W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, Feb 2003.
2. 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.
3. R. Chinnici, M. Gudgin, J. Moreau, J. Schlimmer, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0. W3C Recommendation, 2004. `http://www.w3.org/TR/wsdl20` accessed June 2004.

4. Cpn tools homepage. http://wiki.daimi.au.dk/cpntools/_home.wiki accessed March 2005.
5. 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.
6. 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.
7. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
8. David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
9. 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 June 2004.
10. M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Haase. *Java Messaging Service API Tutorial and Reference*. Addison-Wesley, 2002.
11. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, MA, USA, 2003.
12. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
13. M. Kaddour and L. Pautet. Towards an adaptable message oriented middleware for mobile environments. In *Proceedings of the IEEE 3rd workshop on Applications and Services in Wireless Networks*, Bern, Switzerland, July 2003.
14. Sun Microsystems. Java remote method invocation specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmi-title.html accessed March 2005, 2003.
15. Websphere MQ family. http://www-306.ibm.com/software/integration/wmq/ accessed June 2005.
16. Microsoft Message Queue (MSMQ) Center. http://www.microsoft.com/windows2000/technologies/communications/msmq accessed October 2004.
17. 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.
18. E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
19. 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.
20. M. Snir, S. Otto, D. Walker S. Huss-Lederman, and J. Dongarra. *MPI-The Complete Reference: The MPI Core*. MIT Press, second edition, 1998.
21. J. Thompson. Toolbox: Avoiding a middleware muddle. *IEEE Software*, 14(6):92–98, 1997.