TripCom

Triple Space Communication

FP6 – 027324

Deliverable

D8A.4

EAI Best Practice Guide

Daniel Martin (USTUTT)
Daniel Wutke (USTUTT)

April 1, 2009

Executive Summary

In this report, we compare two classes of middleware systems: tuplespaces as the underlying communication paradigm of TripCom and message oriented middleware. Unlike in related approaches, the comparison is not done on the level of individual features or levels of decoupling, we rather compare these systems by discussing how recurring problems in the area of enterprise application integration can be solved. The discussion therefore is based on "Enterprise Integration Patterns" that are documented in a highly recommended book by EAI practitioners from all over the world. We focus on the patterns that reveal the most substantial differences of both technologies and conclude with a section on EAI "best practices" for tuplespaces – EAI patterns where tuplespaces provide a solution superior to what messaging technology provides today.

Document Information

IST Project Number	FP6 – 027324	**Acronym**	TripCom
Full Title	Triple Space Communication		
Project URL	http://www.tripcom.org/		
Document URL			
EU Project Officer	Werner Janusch		

Deliverable	**Number**	8A.4	**Title**	EAI Best Practice Guide
Work Package	**Number**	8A	**Title**	Use Cases 1 - EAI

Date of Delivery	**Contractual**	M36	**Actual**	31-March-09
Status	version 1.0		final ☒	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	Daniel Martin (USTUTT) Daniel Wutke (USTUTT)			
Resp. Author	Daniel Martin (USTUTT)		**E-mail**	daniel.martin@iaas.uni-stuttgart.de
	Partner	USTUTT	**Phone**	+49 (711) 7816-485

Abstract (for dissemination)	In this report, we compare two classes of middleware systems: tuplespaces as the underlying communication paradigm of TripCom and message oriented middleware. Unlike in related approaches, the comparison is not done on the level of individual features or levels of decoupling, we rather compare these systems by discussing how recurring problems in the area of enterprise application integration can be solved. The discussion therefore is based on "Enterprise Integration Patterns" that are documented in a highly recommended book by EAI practitioners from all over the world. We focus on the patterns that reveal the most substantial differences of both technologies and conclude with a section on EAI "best practices" for tuplespaces – EAI patterns where tuplespaces provide a solution superior to what messaging technology provides today.
Keywords	EAI, Patterns, Space-based Integration, Linda, TripleSpace

Version Log			
Issue Date	**Rev No.**	**Author**	**Change**
2008-05-20	1	Daniel Martin	Document created
2008-05-22	2	Daniel Martin	Initial Outline
2008-06-04	3	Daniel Martin	First text drafts
2008-06-27	4	Daniel Martin	Text revised, new chapter
2008-08-12	5	Daniel Martin	Added sections
2008-08-29	6	Daniel Martin	spell checking, summary table
2008-09-02	7	Daniel Martin	More text on differences
2009-01-22	8	Daniel Martin	Minor changes, proof reading
2009-03-20	9	Daniel Martin	Finalized

Project Consortium Information

Acronym	Partner	Contact
Leopold Franzens University Innsbruck http://www.deri.at	LFUI	Prof. Dr. Dieter Fensel Digital Enterprise Research Institute (DERI) Innsbruck, Austria E-mail: dieter.fensel@deri.org
National University of Ireland, Galway http://www.deri.ie	NUIG	Dr. Laurentiu Vasiliu Digital Enterprise Research Institute (DERI) Galway, Ireland Email: laurentiu.vasiliu@deri.org
University of Stuttgart http://www.iaas.uni-stuttgart.de/	USTUTT	Prof.Dr. Frank Leymann Inst. für Architektur von Anwendungssystemen (IAAS) Stuttgart, Germany E-mail: frank.leymann@informatik.uni-stuttgart.de
Vienna university of Technology http://www.complang.tuwien.ac.at/	TUW	Prof.Dr. eva Kühn Institut für Computersprachen Vienna, Austria E-mail: eva@complang.tuwien.ac.at
Free University Berlin http://www.ag-nbi.de/	FUB	Prof. Dr.-Ing. Robert Tolksdorf AG Netzbasierte Informationssysteme Berlin, Germany E-mail : tolk@inf.fu-berlin.de
Ontotext Lab, Sirma Group Corp. http://www.ontotext.com/	ONTO	Atanas Kiryakov, Vassil Momtchev, Ontotext Lab, Sirma Group Corp. Sofia, Bulgaria E-mail: vassil.momtchev@ontotext.com
Profium OY http://www.profium.com/	Profium	Dr. Janne Saarela Profium OY Espoo, Finnland E-mail: janne.saarela@profium.com
CEFRIEL SCRL. http://www.cefriel.it/	CEFRIEL	Davide Cerri CEFRIEL SCRL. Milano, Italy E-mail: cerri@cefriel.it
Telefonica I+D http://www.tid.es/	TID	Noelia Pérez Crespo Telefonica I+D Madrid, España E-mail: npc@tid.es

Table of Contents

1 Introduction

Enterprise applications consist of a large number of individual applications, often comprising roll-your-own applications, legacy applications, acquired standard applications, etc. running on multiple application servers and depending on various data sources. These applications must be constantly improved and adapted according to business needs. Furthermore, the individual applications are typically built without integration in mind. Because business users of enterprise applications do not use them in isolation but require an integrated and inter-operable seamless use, a separate integration layer must be provided to result in the required business view of the application scenery. Also, enterprise application infrastructures grow and evolve over time, thus, the separate integration layer grows at the same pace. Building such an integration layer is typically referred to as creating an *Enterprise Application Integration* (EAI) solution.

When creating an EAI solution, one often recognizes certain recurring problems and that the same actions are taken over and over again to solve these problems. Recurring problems and their solutions can be described as *Patterns* [1]. Besides the well-established patterns in the field of object-oriented software design [9], there is an emerging area for creating patterns for software and enterprise architecture [5, 3]. [11] defines a set of patterns, covering prominent examples of situations encountered in message-based EAI solutions.

Enterprise application integration, "the unrestricted sharing of data and business processes among any connected application or data sources in the enterprise" [13], refers to a set of technologies that solves two fundamental problems: (i) exchange of data between heterogeneous applications in a reliable, asynchronous and loosely coupled manner [12], (ii) sharing functions between different platforms in an interoperable manner. In practice, EAI solutions are based on *message oriented middleware* (MOM) technology [2].

Tuplespace-based computing, originally based on the concept of *Linda* spaces [10], provides an alternative to messaging middleware as a means for reliable, asynchronous and loosely coupled communication and might therefore provide a suitable platform for EAI solutions. EAI patterns on the other hand have been described assuming MOM as their underlying middleware. In this report, we investigate if existing space-based computing technologies accomplish the requirements originating from EAI patterns.

This report provides a comparison of messaging and triplespace technology by investigating the practical application of both in the field of EAI. For this purpose, we provide a comprehensive analysis of the landscape of EAI patterns presented in [11] by assuming tuplespace technology, the underlying communication paradigm of Triplespaces, as their underpinning. When we have to argue on an implementation level, we assume *Java Message Service (JMS)* [15] or *TripCom*[1] respectively, as implementation of MOM or triplespace technology.

[1] `http://www.tripcom.org/`

2 Related Technology Overview

In this section, we provide an introduction to messaging technology, space-based computing and Enterprise Application Integration patterns as basis for further discussions in this report.

2.1 Messaging Technology

Message oriented middleware is a platform for reliable and asynchronous communication. It is based on the notion of messages, the data objects communicated, and channels, logical addresses to exchange messages [11]: data to be sent is encapsulated as a message and put into a channel from which it is consumed by a receiver. The concept of a channel is realized by MOM by transparently forwarding the message from the sender to the receiver and persistently storing it on every node along this message path ("store-and-forward"). Messages are processed asynchronously, i.e. the message sender can continue its execution as soon as the message has been accepted by the MOM ("send-and-forget"). Messaging systems provide a certain quality of service referred to as "guaranteed delivery": by acknowledging the reception of a message, the MOM guarantees that the message will eventually be delivered to either its intended destination or - in case of errors - to a special destination (e.g. a dead letter channel); under no circumstances it will be lost.

2.2 Tuplespace-based Computing

Tuplespace-based computing (TSC) has its origin in the *Linda coordination language*, defined by [10] as a parallel programming extension for traditional programming languages (e.g. C, Prolog) for the purpose of separating coordination logic from program logic. The Linda concept is built on the notion of a *tuplespace*. A user interacts with the tuplespace by storing and retrieving *tuples* (i.e. an ordered list of typed fields) via a simple interface: tuples can be (i) stored (using the *out* operation), (ii) retrieved destructively (*in*) and (iii) retrieved non-destructively (*rd*). Tuples are retrieved using a template mechanism, e.g. by providing values of a subset of the typed fields of the tuple to be read ("associative addressing"). Matching tuples are selected non-deterministically. Also, concurrent destructive reads are non-deterministic, i.e. the receiver of the tuple is chosen randomly.

2.3 Triplespace Communication

TripCom refers to Triple Space Communication, a new form of network-based communication which replaces message-based point-to-point communication methods typical of Web services with communication via publication in a shared space known as tuplespace. The main technologies involved are tuplespace computing, Web services, and the Semantic Web. Triple Space is the name we give to a type of tuplespace extended to support the publication and exchange of semantic data which is based on the RDF data model, consisting of triples of the form subject-property-object which build statements of knowledge.

It is important to note however that the underlying communication paradigm and thus the "coordination model" of Triplespace is equal to that of tuplespaces. To be of broader, general use, the comparison in this work therefore is not based on concrete implementations, but on the conceptual level of communication systems. Therefore, the concept of tuplespaces, not an implementation based on the tuplespace paradigm is used for the comparison in the following sections.

2.4 Enterprise Application Integration Patterns

EAI patterns document common EAI practices. Relations between patterns solving a more complex problem result in a web of patterns referred to as pattern language. Patterns are grouped into categories which represent different aspects of the same problem domain. Those building blocks are: *Messaging Endpoints*, *Message Construction*, *System Management*, *Messaging Channels*, *Message Transformation*, and *Message Routing*. For example the Message Construction category consists of patterns like *Document Message* and *Command Message*. Patterns can be composed: basic patterns represent senders, receivers, or elemental processing steps of an EAI solution that can be wired together via channels resulting in composed patterns. Channels are realized via MOM technology, thus, messaging is the underlying communication technology for EAI patterns.

3 Discussion of Patterns

In the following sections, we introduce each of the EAI patterns categories mentioned above and discuss in-depth those patterns that reveal the most substantial differences when being implemented on top of TSC middleware rather than on top of MOM. Each section includes a table listing all patterns from the category discussed and a classification according to their significance for differentiating TSC and MOM technology. Patterns listed in column "No Difference" can be implemented on top of an TSC middleware system without major changes. Column "Differentiator" lists those patterns, the implementation of which are either (i) more simple or straightforward, (ii) more complex, or (iii) change their semantics when realized on top of TSC.

3.1 Messaging Endpoints

Messaging Endpoints describe how applications are connected to messaging systems and thus, how applications send and consume messages. Table 3.1 shows an overview of all Messaging Endpoints specified in [11]. Most of the patterns can be applied to TSC without any changes.

Message reception through Message Endpoints can be further detailed by the way messages are received: in case of a *Polling Consumer*, the recipient actively polls the communication infrastructure for data items to consume. In contrast to this, an *Event-driven Consumer* is notified by the communication infrastructure as soon as a new data item becomes available. TSC implements Polling Consumers by providing non-blocking versions of the read primitives that immediately return if no match is found. Event-driven consumers are implemented using notification mechanisms (see e.g. [8]) that allow a consumer to register a template at the middleware system and receive a notification once a matching entry is written to the space.

Selective Consumers consume a subset of messages available on the channel they are listening on. When using Selective Consumers on *Point-to-Point Channels* (see Section 3.4), it is important that each message is taken away from the queue in any case, whether it matches the selection criteria of the consumers or not. Otherwise unconsumed messages would clutter the channel and even make it unusable, e.g. when the channel is configured to follow strict FIFO ordering. Messages not matching the selection criteria of any consumer would remain at the head of the queue, preventing newly arriving messages at the tail from being consumed. This can only be avoided by a careful definition of the consumer's selection criteria to ensure that any possible message on the channel is accepted by at least one of its consumers.

In TSC, the middleware itself already provides functionality equivalent to the Selective Consumer pattern. Clients directly access data by describing the desired content in the form of a query, also known as "template". This way of accessing data is fundamentally different from what is used in messaging technology[1]. Furthermore, the problem of unconsumed messages cluttering a channel does not exist in TSC since there is no specific ordering of data.

Durable Subscription is a pattern for *Publish-Subscribe Channels* where the messaging system retains a separate copy of each message for each subscriber - even if a

[1]Note that JMS [15] defines "Message Selectors" that implement the Selective Consumer pattern in MOM. However, these selectors can only be applied to the header part of a message.

subscriber is not available. When a Durable Subscriber reconnects, the middleware delivers all missed messages in the order specified.

These semantics cannot exactly be reproduced in TSC. If a client disconnects, all data *may* be kept and be still available at reconnection - but the space does not guarantee this. A space naturally keeps all data that was written until someone destructively reads it. In this case the disconnected client is not even aware that there was an object available during the time it was disconnected. Consequently, there is no simple way in TSC to ensure that temporarily disconnected clients know about the history of actions performed on the space (e.g. tuple removal) while they are disconnected. We call this the *incomplete history* problem. The reason for this is that a space does not create a separate copy of a tuple for each subscriber, there is only one single copy of a tuple available in the space until it is explicitly withdrawn.

Table 3.1: Messaging Endpoints Patterns

Pattern Name	**No Difference**	**Differentiator**
Messaging Gateway	X	
Messaging Mapper	X	
Transactional Client	X	
Polling Consumer	X	
Event-driven Consumer	X	
Competing Consumer	X	
Message Dispatcher	X	
Idempotent Receiver	X	
Service activator	X	
Selective Consumer		X
Durable Subscriber		X

3.2 Message Construction

Message construction patterns (see Table 3.2) suggest ways how applications that use *Message Channels* can actually exchange a piece of information. The fundamental pattern of this category is the *Message* pattern, which describes how to structure information in form of a message consisting of a header and a body part. The message header contains information relevant for message delivery (e.g. destination address) and is intended to be interpreted by the communication infrastructure. The body of a message contains the actual payload that is to be delivered to a receiver.

In contrast to messaging technology, TSC does not have the notion of a *message*. Its basic data structure is that of a *tuple*. A tuple can be of arbitrary structure, thus mapping from message to tuple format is straightforward. The problem raised by this procedure is, messaging systems are aware of the semantics of the message header fields and are able to interpret them during message delivery, whereas a space-based

system is not. To simulate the functionality provided by messaging systems in TSC, it would therefore be necessary to create an extra layer of functionality at the application level to interpret the header part of the converted message.

Messages themselves can serve different needs, e.g. in the form of a *Command Message* to invoke remote functionality in an RPC-like manner, in the form of a *Document Message* to transport data between applications and in the form of an *Event Message* to communicate event information from one application to (possibly multiple) others. After addition of message-specific application logic to TSC, all those forms of messages are obviously supported by TSC as well, given the support for messages in general.

Message Expiration shows ways how to attach timeouts to messages to indicate that they are no longer valid and should be considered stale. JMS [15] for example implements this pattern with its "time-to-live" parameter. Similarly, many TSC implementations provide tuple expiry mechanisms, e.g. in the form of a lease (see e.g. [8]).

The *Request-Reply* pattern is a well-known message exchange pattern that is typically built on top of the *Return Address* and *Correlation Identifier* patterns. It uses a Return Address in the request message to tell the receiver where to send the reply to. The Correlation Identifier pattern is used to specify the request corresponding to a certain reply message. In MOM, a channel transmits messages only in a unidirectional manner. Request-reply interaction however requires messages to flow back and forth bidirectionally. This is why messaging APIs often provide a way to create so-called *temporary queues*, point-to-point channels that are only valid for the time of a request-reply cycle. The temporary queue is used as return channel for the replying application.

This is quite different in TSC: spaces do (i) not define the concept of temporary queues and (ii) space-based communication is not bound to the one-way paradigm (see Section 3.4). Request and reply messages can be communicated over the same medium, i.e. the same space. Space-based request-reply communication can e.g. be achieved by adopting the concept of Correlation Identifiers in TSC: when responding to a request tuple, a client would relate the response tuple to the request via the Correlation Identifier, which is also used by the initiator of the request in its template to retrieve the reply tuple.

3.3 Systems Management

Systems management patterns suggest ways how to monitor and control applications and the messaging system itself, how to observe and analyze traffic and how to test and debug distributed MOM-based applications. In this section, we discuss monitoring with the *Detour* pattern and traffic analysis and observation with the *Wire Tap* and *Smart Proxy* patterns. We do not discuss patterns from the testing and debugging section since they are described as strategies that are valid and reasonable for TSC as well. Consequently, these are listed in the "No Changes" column of Table 3.3.

The *Detour* pattern allows to introduce intermediary steps in a messaging application to perform validation, testing or debugging tasks. It is a composite pattern that uses a *Context-based Router* with two output channels where one channel is the direct route to the destination and the other channel contains the inserted processing

Table 3.2: Message Construction Patterns

Pattern Name	No Difference	Differentiator
Command Message		X
Document Message		X
Event Message		X
Message Sequence	X	
Format Indicator	X	
Message Expiration	X	
Message		X
Return Address		X
Correlation Identifier		X
Request-Reply		X

steps, but then leads to the same destination as the direct route. Instructions when to switch between the direct channel and the detour are sent over a *Control Bus* [11].

Implementation of a Detour with space-based middleware is not straightforward. It requires the introduction of a mechanism providing a destructive read operation which is applied *before* the original receiver of the tuple consumes it. This way, a client implementing the Detour could intercept the original message, process it and either write it back to the space so that the original receiver can consume it or modify it in a way that triggers execution of the next inserted processing step. This is however not possible with current space implementations: since the Detour client registers the same template as the original receiver, it has to be guaranteed that upon insertion of a respective tuple, the client implementing the Detour executes first. According to [10], it is not determined which one of a list of clients that query the space for the same template is the one that actually receives the tuple.

An alternative approach to solving this problem in TSC would be to design the application that includes the Detour pattern as if it would be based on a MOM (i.e. using a Context-based router with one incoming and two outgoing message queues) and then replace each queue with a separate space, thus effectively using spaces the same way as queues are used. This requires of course the use of routers between spaces, just as routers are used between message queues. A Detour can now be implemented using the same strategy explained previously - as a context-based router with two outgoing channels.

The *interception problem* as outlined above is raised not only by the Detour pattern, but also by the Wire Tap and Smart Proxy patterns. Common to all of them is the need to intercept a (destructive) read operation and to make sure that it is executed after applying specific processing steps to the original message.

3.4 Messaging Channels

Messaging Channels are used to establish connections between applications and are therefore the fundamental part of a messaging-based application. From the applica-

Table 3.3: Systems Management Patterns

Pattern Name	No Difference	Differentiator
Control Bus	X	
Message Store		X
Detour		X
Message History	X	
Wire Tap		X
Test Message	X	
Smart Proxy		X
Channel Purger	X	

tion's point of view, a channel is a logical address to write messages to and retrieve messages from. According to [11], each channel is associated with an assumption about the particular "sort of messages" communicated over that channel. A channel acts on behalf of the sending and receiving applications and thus provides them a way to communicate anonymously: i.e. the sending application is anonymous to the receiving application and vice versa. Applications choose a channel to communicate messages based on the expectation of the type of messages that run over that channel. Note that applications themselves still communicate anonymously over that channel; they do not know each other, but they share a common assumption about the sort of messages that are communicated.

This is a major difference to the notion of a space. According to [4], a space is a "globally-shared collection of ordered tuples". Applications do not establish connections between each other directly, they rather just write data into or read data from the space. The difference to messaging is that there is no specific intention associated with this action. This is actually the reason why Linda is called a *generative language* [10]. It is emphasised that "a tuple generated by an application A has an independent existence in the TupleSpace until explicitly withdrawn". Further, this means that the concept of a logical address for writers and consumers does not exist in TSC: data is accessed only based on content ("associative addressing" see e.g. [14]) using a matching technique that uses tuples and anti-tuples (very similar to "query by example" [17]).

Another significant difference is the direction of the message flow. Channels are unidirectional [11], all messages transmitted over a channel have the same direction (e.g. from A to B). Having the same direction means that applications are only allowed to do one single operation on a specific queue: either put or get. If the application wants to send a message in the other direction (i.e. a put instead of get to send a response in a two-way communication), a separate channel is required. The reason for this is that if channels where bidirectional, applications that write and immediately read a message from the same channel are very likely to consume the same message that they just wrote to the channel. Making channels unidirectional is a simple, yet elegant way to avoid such situations. In TSC however, such solutions are not even necessary: spaces do not have the concept of delivery. Tuples are rather just stored into a space, whereas a store operation essentially can be seen as a publication of data

which does not imply a direction. It is of course possible that one can consume the same tuple that was written previously, but this would be intentionally because there is no FIFO ordering of tuples in TSC and the read operation must have specified a query that matches exactly the tuple to be read.

Table 3.4: Messaging Channel Patterns

Pattern Name	**No Difference**	**Differentiator**
Invalid Message Channel	X	
Guaranteed Delivery	X	
Channel Adapter	X	
Messaging Bridge	X	
Message Channel		X
Dead Letter Channel		X
Point-to-point Channel		X
Publish-Subscribe Channel		X
Datatype channel		X
Message Bus	X	

Dead Letter Channel and *Invalid Message Channel* are patterns that suggest solutions how to deal with messages that can not be delivered to a destination or that can not be processed at the endpoint itself. The Invalid Message channel is used by a consuming application when a received message cannot be processed (e.g. due to a parse error): this message is then kept at a common place - the Invalid Message Channel - for later analysis. The Dead Letter Channel is used by the messaging middleware itself for storing messages that cannot be delivered. Typically each machine a messaging system is installed on therefore has its own Dead Letter Channel. This channel contains valuable information for system administrators: they can see which machine was the last on the path the message took on its way to the ultimate receiver. An error therefore very likely is between this machine and the next hop on the message's planned path.

Unlike most of today's messaging systems (e.g. WebsphereMQ), TSC middleware does not have a built-in support for Dead Letter Channels. However, as spaces can be deployed in a distributed manner, the concept of a Dead Letter Channel could be a valuable addition to space-based middleware. Just as in messaging technology, one could envision a dedicated space per middleware instance that collects tuples the system was unable to deliver. The Invalid Message Channel pattern can be applied to TSC similarly as it is used in messaging: application code detects non-processable messages and then moves them to an appropriate place.

Point-to-Point Channels are an extension of the *Message Channel* pattern discussed above to facilitate point-to-point interaction over messaging technology. Their main purpose is to ensure that a message sent over a channel is consumed by exactly

one receiver. Messaging channels in general allow multiple receivers to listen on the same channel; instead of allowing just one receiver, the channel ensures that only one amongst the list of all receivers can get a particular message. This behavior is especially interesting to enable remote method calls (RPC) over messaging technology. Using a Point-to-Point Channel, the caller can be sure that a method is called exactly once per invocation request. Note that since point-to-point communication is such an important concept they are first class citizens in a number of implementations, and are explicitly exposed via a specific interface, e.g. as Queues in JMS.

Although there is no explicit notion of a channel in TSC as discussed previously in this section, spaces still provide a way to allow point-to-point interaction. It comes in the form of a destructive read operation (sometimes also referred to as *take* operation). This operation ensures that the first to read a tuple removes it as part of the read operation. If there are multiple receivers requesting the same tuple, the space middleware non-deterministically selects [10] one amongst them.

The *Publish-Subscribe* pattern in messaging is the exact counterpart of the Point-to-Point Channel described previously - it specifically allows multiple receivers to consume the same message. The underlying idea is very similar to the Observer design pattern from [9]. Subscribers can express their interest in certain events and get notified by the publisher *exactly once* while connected, as soon as an event matching the criteria they have subscribed to occurs. As *all* subscribers currently listening on the same topic have to be notified, an event can only be deleted from the messaging system once all subscribers received the notification. This logic should be implemented by the middleware itself releasing subscribers from the burden of coordinating their work to make sure everyone of them was notified and thus allowing them to co-exist independently from each other. A publish-subscribe channel accomplishes these tasks by virtually creating a channel per subscriber and allowing only one consumer per channel. The messaging middleware creates multiple copies of a message sent to a pub-sub channel and sends these copies to each of the subscription channels. This guarantees that every subscriber receives the desired messages and (since it is the only subscriber on the channel) receives it exactly once. If a subscriber goes offline, messages delivered during that time are lost unless a *Durable Subscription* is used.

These specific semantics are hard to achieve in space-based systems. Tuples can obviously be (non-destructively) read by multiple parties, but the challenge is to decide when to remove a tuple from the space. TSC middleware does not have any information about how many clients are interested in a specific tuple and when the last interested party has finally read it. Tuple removal in this case is up to the clients itself. A major drawback of this solution is that it requires direct coordination between interested parties and therefore introduces tight coupling between them.

Another problem is to ensure that a specific party receives a tuple exactly once. Messaging middleware solves this by virtually creating a channel per subscriber and sending a copy of the message to each of those channels. A space-based system in contrast does not create copies, thus leaving this task again to client applications.

4 Best Practices

Table 4.1 summarizes key findings from the discussions in the previous section.

4.1 Tuple vs. Message

The first major difference we found is the mismatch of data objects native to each technology. While TSC uses tuples, MOM expects messages to be communicated. It is not a syntactical mismatch however, since tuples and messages can be arbitrarily structured and mapping between both formats is straight-forward. The mismatch rather is on a semantical level: messages contain two designated parts, message header and message body. The header part contains information that is addressed to the MOM itself, whereas the body of a message is addressed to the ultimate receiver. Tuples do not distinguish between header and body, thus information targeted at the transmission middleware is intermixed with data targeted at the ultimate receiver.

4.2 Implicit Routing

A result of Linda's generative communication paradigm (see Section 3.4) and the absence of (directed) channels is, that tuples exist in the space independently of any client application. They do not have a designated message path and they can be accessed in the same way by any client that has access to the space. This means that e.g. the publisher of a tuple can decide to retrieve this tuple, modify it, and write it immediately back to the space. This way of interaction with a space is not supported in MOM based application systems: a message, once sent to a channel, can not be retrieved by anybody else than the client consuming messages from the channel. Moreover, messaging channels are uni-directional, meaning that the client that sends a Message to a channel cannot itself be a consumer on that same channel.

Messaging uses explicitly defined Message Channels to deliver messages from a sender to possibly multiple receivers. As applications that build upon messaging do typically consist of a network of various message channels (point-to-point, publish-subscribe), message routers are necessary to move messages from one channel to other channels based on information in either the message header or the message content. In TSC, messages are not delivered from a sender to a consumer directly, but rather published to a shared data space by the sender and retrieved by the consumer via a query-by-example technique called template. That means that a routing process with explicit routing rules as it is in messaging technology doesn't exist. Routing in TSC is the internal process how a template is resolved (e.g. using an index) and how the resulting tuples are handed back to the client. Also, TSC is centered around the idea of a "shared data space", meaning that even the data flow itself – explicitly defined by queues in messaging technology – are not visible anymore and again are defined in an ad hoc manner by the query template.

4.3 The Benefit of Semantics in EAI

Traditional EAI solutions stipulate a common data format to avoid syntax transformation between heterogeneous data formats and syntaxes. There is a whole range

Table 4.1: Most Significant Differences of Spaces and Messaging

Differentiator	**TSC**	**MOM**
Tuple vs. Message	Not aware of the semantics of a Message.	Aware of the semantics of a Message.
"Implicit Routing"	Spaces do not have the concept of routing.	Message routes are explicitly defined at design time.
"Interception Problem"	It is not possible to define the order of concurrent read operations on the same tuple	Since all possible routes for messages are defined explicitly, this problem does not exist.
Original Queue/Publish-Subscribe Semantics	Not exactly reproducible with TSC	Naturally supported by MOM technology.
"Incomplete History"	A client may miss interesting messages when being offline.	The Durable Subscription pattern allows clients to be temporary offline, but still ensures that messages are re-delivered when coming online again.

of possible data formats and standards to chose from for this purpose. The most widely used Electronic Data Interchange (EDI) systems are EDIFACT (a UN recommendation predominant in Europe) and its U.S. counterpart X.12. Both systems were originally based on ASCII data formats, but now have XML serializations as well. Numerous newer systems based on native XML have been created with RosettaNet and ebXML being two of the most popular. The challenge with XML-based formats is that merging of messages requires *a priori* schema knowledge and custom-built XSL transformations. Knowledge of the applied schemas is particularly problematic when operating in open environments such as the Web, which are characterized by heterogeneous data formats and vocabularies.

As the common data format for exchanging messages, TripleSpace in contrast uses RDF [16], which utilizes a data model based on graphs. RDF allows to denote objects via URIs, and describe the connections between them. The main benefit of using RDF is the ability to share concepts across organizational boundaries. E.g. it is possible to use Wikipedia URIs to denote the category of content that is being requested. This way, it is possible to share data items without too much previous agreement across the actors, which enables ad-hoc formation of workflows and data exchange.

Another benefit of using RDF is the ability to merge RDF documents from a number of sources without the need for schema integrating at the merging step. Similarly, it is possible to retrieve only parts of the graph (unlike retrieving the whole message in MOM) or combining fragments of other graphs using a sophisticated query and graph construction language such as SPARQL.

Finally, the ability to express facts and logical relationships between them may prove useful in certain EAI scenarios. Knowledge inference can be expressed in a

generic manner that is executed directly in the middleware, and is not generated by custom application logic that runs as client to a MOM.

5 Conclusion

This report compares MOM and TSC middleware technology by investigating the differences that arise from using TSC as underlying communication technology for solving common EAI problems.

While it has been found in related work that TSC and MOM are similar with respect to the degree of decoupling they offer to communicating partners, this work shows that their inherent concepts are very different. The most substantial differences discovered in this report are:

- Messages in MOM travel through explicitly defined routes whereas routing in TSC is "implicit".
- While the syntactic mapping between messages and tuples is straightforward, the difference is on the semantic level. Tuples do not distinguish between header and body, i.e. between data addressed to the middleware system and to the ultimate receiver.
- TSC does not have an equivalent counterpart of a Durable Subscription: clients in TSC may miss tuples they are interested in while being offline, their "historical knowledge" about the system is incomplete.
- TSC does not allow to define the order of concurrent read operations on the same tuple, a fundamental requirement for e.g. the Detour or Wire Tap pattern that intercept the message flow.
- The original Queue and Publish-Subscribe semantics from MOM are not exactly reproducible in TSC, but naturally supported by message-oriented middleware.
- The use of a common, generic data format such as RDF promises to be of high value, e.g. in scenarios where data heterogeneity and mediation is required.

Based on this, we argue that TSC is not a suitable drop-in replacement for MOM in EAI scenarios. On the other hand, TSC offers a fundamentally different approach to the exchange of data. Instead of explicitly routing messages to recipients, data is published to and associatively retrieved from a shared space. For EAI scenarios that require open and ad-hoc collaborations between distributed parties, with possibly different data schemas used on each partner's side, this offers new possibilities which are investigated in e.g. [7] and [6].

References

[1] C. Alexander et al. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] B. Blakely, H. Harris, and L. Rhys. *Messaging and Queueing Using the MQI: Concepts & Analysis, Design & Development*. McGraw-Hill, 1995.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. Wiley, 1996.

[4] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems (TOCS)*, 4(2):110–129, 1986.

[5] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[6] D. de Francisco, J. M. Elicegui, D. Martin, M. Murth, and D. Wutke. Using Triple Spaces to Implement a Marketplace Pattern. *In proceedings of the 1st Workshop for Space Based Computing as Semantic Middleware for Enterprise Application Integration (SBC 2007), Vienna, Austria*, 2007.

[7] D. de Francisco, N. Perez, D. Foxvog, A. Harth, D. Martin, D. Wutke, M. Murth, and E. Paslaru. Towards a Digital Content Services Design Based on Triple Space. *In proceedings of the 10th International Conference on Business Information Systems (BIS), Poznan, Poland 25-27 April*, 2007.

[8] E. Freeman, K. Arnold, and S. Hupfer. *Javaspaces (tm) Principles, Patterns, and Practice: Principles, Patterns and Practices*. Addison-Wesley Professional, 1999.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[10] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[11] G. Hohpe, B. Woolf, and K. Brown. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[12] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.

[13] David S. Linthicum. *Enterprise Application Integration*. Addison-Wesley Professional, 1999.

[14] R. Menezes, I. Merrick, and A. Wood. Coordination in a Content-Addressable Web. *Autonomous Agents and Multi-Agent Systems*, 2(3):287–301, 1999.

[15] R. Monson-Haefel and D. Chappell. *Java Message Service*. O'Reilly, 2000.

[16] W3C. Rdf primer. W3C Recommondation 10 February, `http://www.w3.org/TR/rdf-primer/`, 2004.

[17] M.M. Zloof. Query-by-Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.