Loosely coupled services with JMS and JavaSpaces

Sören Haubrock

soeren-nils.haubrock@hpi.uni-potsdam.de

Loose coupling is a central aspect in service-oriented computing. While the general concept of service-oriented architectures (SOAs) does not imply any particular restrictions on how to realize this aim, certain technologies provide ways to support loose coupling and thereby follow different approaches.

In the Java world, two technologies are particularly aiming at providing middleware components to support message and workload exchange in an asynchronous fashion. While the *Java Message System (JMS)* follows the idea of a central platform for message exchange in order to de-couple the communicating partners, the *JavaSpaces* approach relies on the concept of a space-based object and data exchange repository in order to distribute workloads among several participants in a highly-flexible way.

In the scope of this paper, these two approaches are described, exemplified and analysed with respect to their role in service-oriented computing.

1 Loose coupling in service-oriented computing

The concept of a service-oriented architecture (SOA) relies on a set of certain aspects, which can be summarised by the definition of Wilkes [1]: “[A SOA is] a way of designing and implementing enterprise applications that deals with the intercommunication of **loosely coupled**, coarse grained (business level), reusable artefacts (services) that are accessed through well-defined, platform independent, interface contracts.“

Loose coupling is a fundamental aim in service-oriented computing. „It describes a resilient relationship between two or more computer systems that are exchanging data. […] with few assumptions about the other end“[3].

The benefits of loose coupling are seen in the transparency, flexibility in large application composition and resource binding[4,5]. Hereby, the aspects of de-coupling in terms of location, time and reference at build-time are to be differentiated.

De-coupling in terms of reference is already realised in the SOA-implementation of SOAP web services. While the interface of such a service needs to be known a-priori (or at least to be found and understood), the actual instance of a service can be deployed and exchanged at run-time.

In the same case, the location of a web service is de-coupled from the client in the sense that it can be looked up at run-time from a registry. The location of a service itself might change from one URL to another as long as the registry consistently updates the location changes. However, once a consumer of a web service does not use the registry anymore (to find the service) or the address in the WSDL document is not up-to-date, the invocation of a service leads to malfunction. Therefore, the realisation of space-de-coupling in SOAP web service implementations strongly depends on the policy of both, the service provider and the service consumer.

Regarding the time aspect, the current standard SOAP web services do not provide a means to systematically de-couple communicating end points from each other. However, only when components are de-coupled not only by reference, but also by location and time, the establishment of SOAs with asynchronous information exchange becomes feasible. As an example, memory- and time-consuming processes encapsulated by services might run in the background after being invoked, while a client disconnects from the service and proceeds with other tasks without waiting for the service to be finished. Later on, the client may fetch the result data either by asking for the state of a service process or by being informed through the service or some other component after the completion of the job. Both approaches sketched in this paper realise time-decoupling in a specific way.

2 Java Message Service (JMS)

2.1 Loose coupling in messaging systems

In messaging systems, applications are loosely coupled through the exchange of self-describing messages. In theory, these messages can contain any information needed by the implicit communication protocol between sender and receiver, i.e. some text information, an XML-document (e.g. SOAP message) or binary data.

In such a system, a component is needed to receive, store and send messages for all participating endpoints. In most cases, this functionality is provided by a service, which can be accessed by the communication partners. The components responsible for the messaging process including these access services are subsumed as Message Oriented Middleware (MOM). Obviously, there are certain requirements to be fulfilled by such a messaging system in terms of reliability, accessibility, security and performance.

With the message service as a central component in this architecture, communication partners in a messaging architecture are de-coupled in all relevant aspects. The endpoints do not communicate directly with each other, but rather send messages to the service. Thus, each communication partner only needs to know the reference and location of the message service itself.

With respect to time, the messaging approach provides a highly flexible way to realise de-coupled communication. After sending a message to a messaging system, it either remains there until the (set of) client(s) fetch(es) it, or a certain time-out policy prevents the server from being overloaded.

2.2 What is JMS?

The Java Message Service API is a messaging standard that allows application components based on the J2EE platform to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous [6].

Existing messaging systems, such as *Bea WebLogic*, *IBM MQSeries* or *MSMQ*, each have their specific, non-standardised interface and are therefore not compliant with one another. JMS provides a standard Java-based interface to these messaging systems. The idea of the Java Message Service is to provide a message transport service, called the provider, implementing the JMS interface and giving access to the proprietary system.

The specification does not define how messages need to be transported within a particular implementation. This clear separation of concerns was essential in order to allow vendors of existing messaging products, to support the JMS specification.

2.3 JMS messaging approaches

In the JMS context, the co-called endpoints play an important role. Instead of sending a message directly to its receiver address (tight coupling), the sender delivers the data to an abstract endpoint, which takes care of the following tasks.

On the other side, the receiver to whom the message is dedicated, does not fetch it from the sender, but rather determines, from which endpoint it is willed to receive messages. It is therefore possible, that multiple receivers consume messages from the same sender, as long as they all determine the same endpoint to communicate with. The communicating partners do not need to “know” about each other.

At the same time, the receiver might fetch messages from multiple senders. This is the case, when each of the senders puts its messages to an endpoint that the receiver communicates with.

The two implementations of the different messaging approaches are described in the following: p2p and publish/subscribe messaging.

2.3.1 Point-to-point messaging (p2p)

Messages are most often dedicated to one specific receiver. In this case, the task of a messaging system is to provide a storage capacity accessible via endpoints that receive messages from the sender and store them in a queue, meaning that a message fetched from the queue (consumed) is gone and therefore not accessible by other consumers anymore. The message system schedules the message forwarding in a FIFO order, the next query for a message at the same endpoint serves the following message. Figure 1 shows the p2p approach for a simple set-up of two message exchanging communication partners. In figure 2, the case of two receivers fetching from the same queue is depicted exemplary.

The two core characteristics of this concept are on the one hand the fact that each message can be consumed at most once due to the queue storage system. On the other hand, the dynamic addition of senders and receivers to the infrastructure at runtime offers a high flexibility, fulfilling some important criteria of service-oriented architectures.

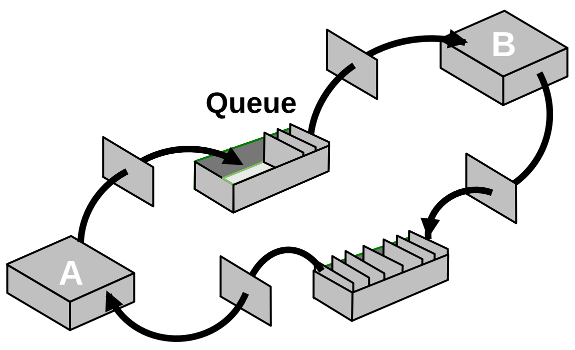

Figure 1: p2p-Messaging architecture [8]

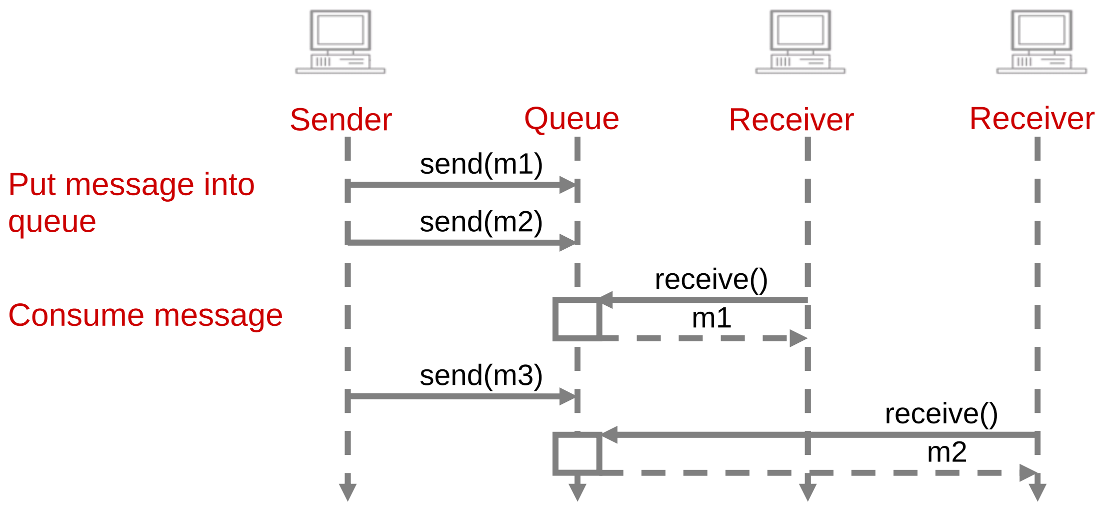

Figure 2: Message sequence in p2p-messaging [8]

While the JMS does not explicitly interdict that multiple consumers share the same endpoint for receiving messages to be read at most once from a single sender, the p2p messaging approach is not intended for this communication pattern. JMS does not specify any semantics for this case either.

2.3.2 publish/subscribe messaging (pub/sub)

For the case that many consumers are interested in the same information from a sender, the second communication pattern can be used.

Instead of providing queues for the messages, the endpoints are representing so-called topics, to which consumers are able to subscribe. Similar to the concept of a mailing list, each communication partner interested in a certain topic will receive the messages that are bound to the specific endpoint. These messages are provided by a publisher of the topic. In general, the communication takes place between m publishers and n subscribers via a single topic (endpoint). Figure 3 shows an example of a single publisher sending messages to an endpoint, to which several consumers are subscribed.

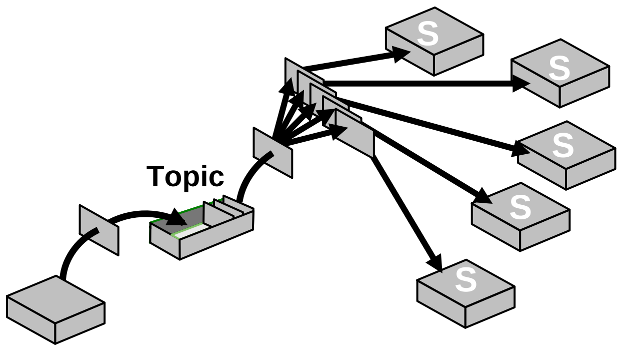

Figure 3: publish/subscribe-messaging architecture [8]

In contrast to p2p messaging, consumers obtain messages without having to explicit request them. Once subscribed to the topic, the system pushes the message to the client as soon as it is accessible and the delivery is being scheduled.

In this approach, it is guaranteed that each subscriber receives a copy of every message belonging to the topic of the very endpoint. However, certain restrictions apply to this rule of thumb, taking into account that accessibility of the consumers and storage capacity of the message system are limited.

2.4 Concepts and application of the JMS API

While Sun developed two different interface families for the messaging architectures in the first API version (JMS 1.0), they have been unified in the version 1.1 from 2002. Table 1 shows these interfaces and their specific names bound to each of the two messaging architectures.

Table 1: Relationship of p2p and pub/sub interfaces [6]

JMS Common Interfaces	P2p-specific Interfaces	Pub/sub-specific interfaces
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
Destination	Queue	Topic
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

The ConnectionFactory is used by clients to set-up a connection with the JMS provider, resulting in a Connection object. A connection can contain multiple sessions, each of which stands for a single-threaded context for sending and receiving messages.

The destination object encapsulates the identity of a message destination (the consumer(s)). The MessageProducer object is created by a Session and used for sending messages to a destination, while the MessageConsumer is used for receiving messages that are sent to a certain destination.

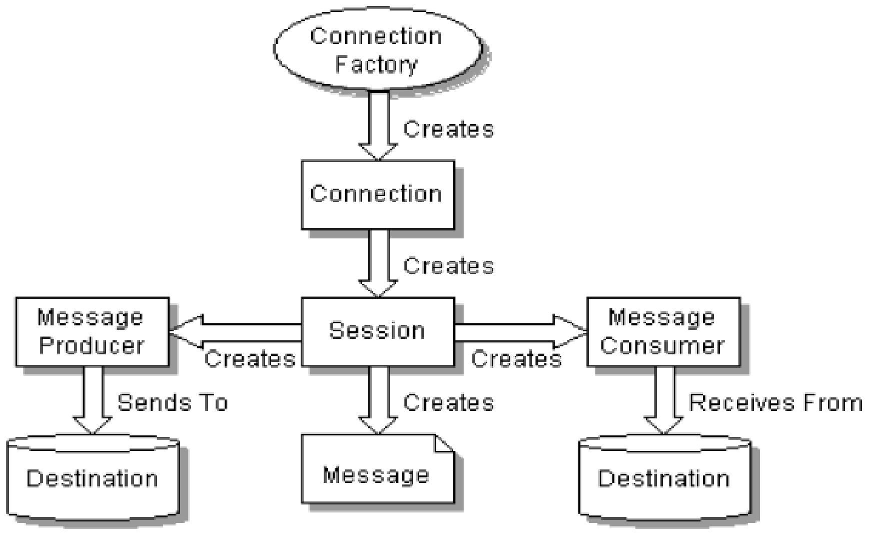

Figure 4: JMS object relationships [6].

A working JMS application consists basically of one or more JMS clients exchanging messages, and the messaging system providing the JMS interface. The following scenario depicts a typical use case where a client consumes a message from a topic endpoint via JMS.

The message consumer looks up a topic connection factory in the JNDI context. The sole purpose of the factory is to create JMS connections. Before it is bound to JNDI, connection parameters can be configured (IP, port, protocol, reconnection and load-balancing strategy). A Topic object, providing a handle for the physical implementation of a topic, can be referenced via JNDI as well.

A TopicConnection is a unique, direct connection to the JMS provider and serves as a factory for TopicSessions. As each connection can potentially mean a pool of threads, an underlying TCP connection and more administrative overhead, in general each client is supposed to have only one connection to a provider.

```
TopicConnectionFactory tcf =
(TopicConnectionFactory)context.lookup("TopicConnectionFactory");

Topic topic = (Topic)context.lookup("Quotes");

TopicConnection tc = tcf.createTopicConnection();
```

With a connection established, a session is created in the next step. The parameters for session creation determine whether a message transfer is transactional and acknowledged, respectively. Furthermore, a session can be used as a factory for messages, topic subscribers and topic publishers. Sessions are not thread-safe.

```
TopicSession ts =
  tc.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
```

Once the session object is created, it is possible to create its *TopicPublisher* object and start the session. Messages can then be sent to the topic with the *publish*-method.

```
TopicPublisher tp = ts.createPublisher(topic);
tc.start();

String quoteStr = "HPIStocks, 23.31";
TextMessage quote = ts.createTextMessage(quoteStr);
tp.publish(quote);
```

On the other side of the communication, the subscriber object is created and can be used in two different forms. With the explicit call of the *receive*-method, messages are fetched synchronously. As an alternative, the listener concept can be utilized in order to inform subscribers automatically when a new message arrives for a certain topic. A *MessageListener* object is used for events of this type, while objects of type *ErrorListener* are used to handle exceptions that might occur (typically security exception or transport failure).

```
TopicSubscriber tSub = ts.createSubscriber(topic);

//synchronous message fetch
Message msg = tSub.receive();
String message = ((TextMessage)msg).getText();

//asynchronous message fetch
tSub.setMessageListener(new MyMessageListener());
tc.setExceptionListener(new ErrorListener());
```

2.5 Application examples of JMS

JMS is a technology providing access to message based systems for the Java environment. With its potential to provide asynchronous communication to exchange messages (text, objects, binary), it is suitable for applications where the context consists of heterogeneous environments with de-coupled systems communicating over system boundaries. Further, JMS is a feasible approach when no immediate response is required from the service, guaranteed delivery of the service request may be necessary, sending or receiving the message may be part of a transaction and security such as authentication and authorization is an option.

Due to de-coupling with respect to time, reference and location, information providers need not be aware of single consumers.

A typical application domain is the general data exchange in user groups, forums, discussion boards or chats. Each of these communication forms needs a certain set of messages to be stored, made accessible and potentially being access restricted to certain roles.

Another application context can be seen in systems where users work on the same data basis and modifications need to trigger notification procedures so the users can synchronise. Typical examples are project management or calendar systems.

A third domain for this technology can be found in the mobile application context, where message exchange heavily relies on message storage capacities working on a 24/7 basis. Hereby, both messaging approaches can be useful. While p2p messaging enables communication between two partners, publish/subscribe is feasible for applications where users are interested in certain topics and want to be informed e.g. by SMS.

3 JavaSpaces

In distributed computing, one approach for a client-server system is based on the use of a virtual compute server implemented on a number of co-operating workstations or PCs. Hereby, clients send requests (jobs) to the virtual compute server, where one or more of its processors complete the job. Since the computational power of the server derives from an aggregation of machines, it is possible to scale the system by changing the number of machines without affecting ongoing activities. In theory, the server can be expanded on the fly to meet peaks in demand by adding machines temporarily. Due to the distribution of processing power, the system can be up and running full time, with hardware maintenance and updates handled incrementally, a few machines at a time.

The overall idea of such a system is to perform a number of jobs simultaneously in a reliable and flexible way at locations that may be physically dispersed. In the context of service-oriented computing, this approach is particularly interesting when processes need to be distributed due to high demands on processing power.

3.1 Virtual shared memory and the space concept

A virtual shared memory is a shared object repository that can be used to store data, which is shared among the components of a distributed program. It is virtual in the sense that no physically-shared memory is required to create it. Its potentials in load balancing, high performance and fault tolerance are seen as the key advantages of such a system [12].

Tuple spaces are one way to implement virtual shared memory. A tuple is a simple vector of typed values (field). Each field may have one of the three basic forms: a constant, an expression that evaluates to a constant or a formal parameter. The tuple space provides a repository of tuples that can be accessed concurrently. Producers post their data as tuples in the space, and the consumers then retrieve data from the space through a certain pattern matching approach. Thus, the tuple space realizes a logical associative memory.

The tuple space concept originates from the Linda parallel programming language, developed by David Gelernter and Nicholas Carriero at Yale University [13]. Linda is implemented as an extension of other (sequential) languages [14]. It consists fundamentally of four operations through which the tuples can be added, retrieved or destructively retrieved from the tuple space.

With the concept of virtual shared memory in form of tuple spaces, it becomes possible to communicate asynchronously and anonymously in a distributed and persistent way.

3.2 What are JavaSpaces?

The JavaSpaces technology is an implementation of the tuple space concept in the Java programming language based on the JINI architecture. It can therefore be seen as a distributed programming model as well as an API [10]. The tuple concept is realized by providing a space for storing regular Java objects with its data and functionality in it.

There are three access operations for objects in the space: write, read and take. With read, a copy of an object is created and can be further modified. With take, the object is not only read, but also removed from the space. Figure 5 illustrates the different access methods.

Space objects are read-only, i.e. the content of the objects can not be modified by users while accessing the space. In order to do so, the object needs to be removed (taken) from the space and written back to it in a modified state. In JavaSpaces it is possible to register a listener for a certain object class or value range in the form of templates by using the *notify* method.

The associative look-up in JavaSpaces is realized by the use of template objects that are passed to the space in order to match with the objects in it.

A template object is passed as a parameter in the read method. Matching between the template object and objects in the space is successful in the case that a) the template is of the same type and b) all values are identical to those in the matched object in space, where null serves as a wildcard. As a precondition for the matching concept to work, all object fields used for matching must be public, non-static, non-final, non-transient and instances of a class (no simple data types).

All objects that are put to the space need to implement a certain interface called *Entry.* Apart from that, objects can be of arbitrary class, state and size.

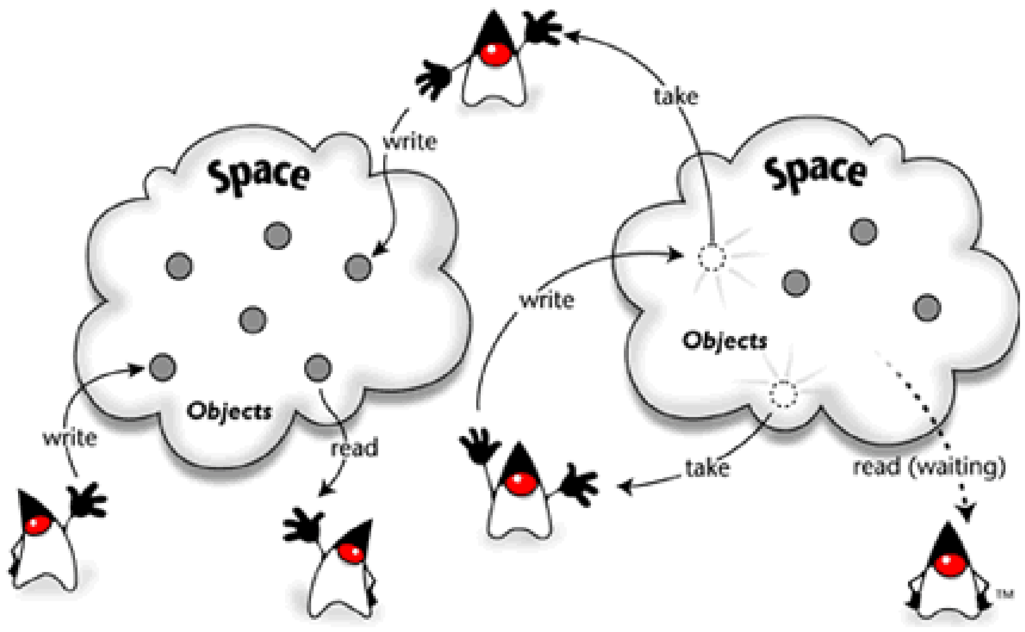

Figure 5: Overview of the JavaSpace concept [9].

A great potential in the JavaSpaces technology is the possibility to store objects together with their functionality in one or several spaces that can be arbitrarily distributed and only need to be made accessible to the clients (directly or indirectly). Thus, workload can be encapsulated in these objects with class-specific implementations. As a consequence, it is possible to make use of processing machines that are unaware of any processing algorithms simply by loading an object from the space, executing a well-defined processing (e.g. *compute()*) method and writing the result back to the space. With the aid of the notification system, clients interested in the outcome of the processing get informed and can then proceed with their work. This concept is called *master*/*worker*-scheme.

3.3 Architecture of the space

The virtual shared memory in form of as space is created by connecting the memories of each participant. Distributed data structures are created by putting objects into space. The space engine replicates those objects to participants that expressed interest. In other words, a distributed cache facility is incorporated into the space infrastructure. Protocols like RMI (or others) are used for the actual transportation. An embedded mode may be also available if participants share the same JVM, in order to avoid the overhead of network serialization.

3.4 Features and limitations of JavaSpaces

JavaSpaces can be run in as a persistent space that maintains state between executions or as a transient space that loses its state between executions. The underlying data representation within the JavaSpace is the serialized version of an object. Figure 6 shows how a JavaSpace operation uses a local proxy to transparently serialize and deserialize entries, based on the principles applied in Jini. The space stores entries in their serialized form and the read or take operations match serialized templates to serialized entries field by field. With each read or write method call, at most one object is processed. Due to the associative lookup procedure, the exact instance to be read can only be determined when the tuples of field values are unique and each tuple value is known by the reader. This is however in most cases not a useful pre-condition given that de-coupling and information hiding of the objects' states is intended.

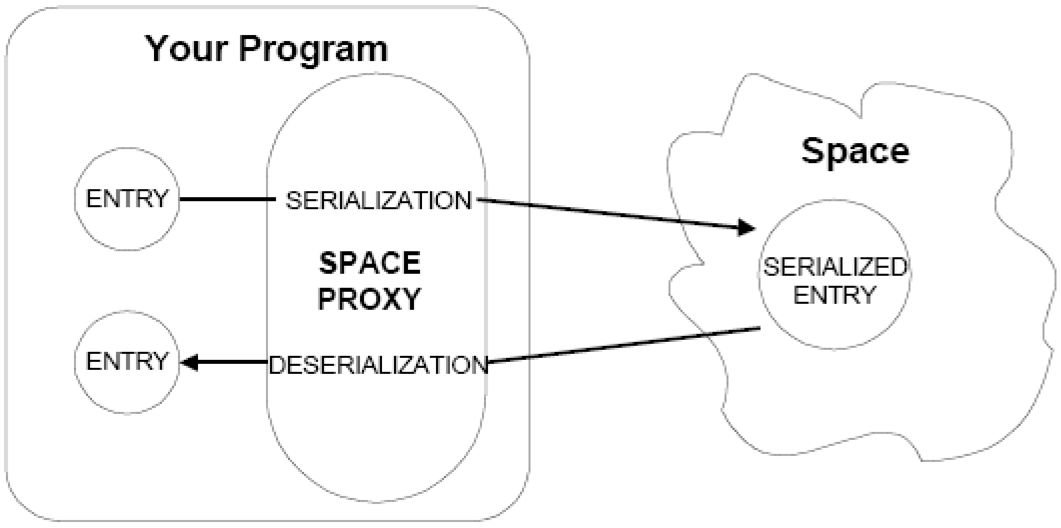

Figure 6: Serialization of entries by a local proxy object before transmittance in the remote space [9].

Obviously, the serialization of objects is a potential bottleneck in the performance of the overall JavaSpaces architecture, especially when being configured as a persistent space. Thus, in order to avoid certain issues such as the network communications and serialization overhead, the query for objects should me minimized where possible.

The literature describing this technology generally promotes the flexibility, expressiveness and simplicity of the framework. However, a number of potential exist, including the inability to extract more than one object at a time and the unpredictable selection of the entries returned. A more fundamental limitation may be the Java centric nature of the service they provide. For a JavaSpace to be suitable for a particular application, the objects that need to be stored persistently must be Java objects. Further, the often asserted scalability is restricted to a certain amount, since local memories are used for storage of serialized objects.

3.5 JavaSpaces example

The described scenario above is sketched in the following listing. It implements the so-called master/worker scheme with processing results written back to space.

Imagine a scenario where image processing (like filtering or geo-referencing) is distributed by using the space as a gateway for distributing the processing burden. For doing so, the image is a-priory sub-divided into a number of smaller image tiles. Each *TileImageProcess* object then contains one of these tiles as a set of data fields (e.g. bytes) and the *compute* method knowing how to process the data.

```
public class TileImageProcess implements Entry {
   public Byte[] imageData;
   public HugeImageProcess() {};
   public HugeImageProcess(Byte[] image) { imageData = image; };
   public void compute();
   public Boolean finished() = false;
}
```

With a static method call to the utility class *SpaceAccessor*, the space can be referenced. This abstraction class (not shown here) allows for hiding the concrete access method. Behind the scenes, JavaSpace objects can be referenced via Jini lookup services or RMI registries.

The additional arguments for the write method determine the transaction procedure used and the lease time of the object, i.e. the time an object is allowed to remain in the space before it will be removed.

```
JavaSpace space = SpaceAccessor.getSpace();
space.write( new HugeImageProcess(imgTile1), null, Lease.Forever);
space.write( new HugeImageProcess(imgTile2), null, Lease.Forever);
space.write( new HugeImageProcess(imgTile3), null, Lease.Forever);
[…]
```

The space is then accessible from the “working services”, so the objects can be fetched from the space and methods executed on high-performance platforms. With the template {*null*, *false*}, only those objects are taken from the space for processing, where the appropriate flag indicates that no processing has taken place so far.

```
HugeImageProcess template = new HugeImageProcess(null, false);

HugeImageProcess todo = (HugeImageProcess)space.read(template, null,
Long.MAX_VALUE)
```

When the processing has finished for an object, the finished-flag is set to *true* and the object is written back to space.

```
todo.finished = true;
JavaSpace space = SpaceAccessor.getSpace();
space.write( todo, null, Lease.Forever);
```

In the case that notification is configured, the client of the image processing gets informed about the finished processing with the following template.

```
HugeImageProcess finishedTpl = new HugeImageProcess(null, true);
```

3.6 JavaSpaces application context

The space functions as a "shared object pool" for communication and coordination that holds state information in a persistent and long-term manner across different communication contexts. Application design patterns are free to choose in which way objects are created and consumed. Instead of publishing interfaces, the space objects are functioning as a bulletin board and implicitly bi-directional communication channel. It's a data-oriented approach creates the illusion of a single address space where the code looks like it's running on a single machine.

Especially its potential in flexible workload-distribution, scalability (up to a certain point) and reliability make it feasible for a range of application classes, where the combination of distributed caching and easy distribution of load is relevant. Typical examples are workflow systems, parallel computing servers, or collaborative systems. In the context of SOA, JavaSpaces provide a valuable component “behind” existing services.

4 JMS and JavaSpaces in SOC

JMS and JavaSpaces both provide a concept and technology that aims at the provision of loosely-coupled communication in terms of reference, location and time. Both approaches provide run-time extensibility, time (store-and-forward) and location independence as well as latency hiding (through asynchronous communication).

JMS is designed for information delivery, whereas JavaSpaces can be called an information-sharing infrastructure. While the JMS approach focuses on the exchange of messages in form of (structured) text or binary data via message-oriented middleware, the JavaSpaces technology is based on the exchange of Java objects via the virtual shared memory.

4.1 JMS in SOC

In a JMS-based distributed system, external clients (or services) use the technology to exchange messages between each other asynchronously. In the context of service-oriented computing, the JMS technology can be understood as a component in the service bus [2]. The actual service calls in a SOA might be maintained, controlled and scheduled by the JMS, providing a layer between the user and the actual services. With this extension, an asynchronous communication can be realised in a more convenient way for the user.

Java Message Services provide a message-centric exchange of small data chunks. In order to make use of the messages, their format needs to be part of a contract between users. In other words, although the communication between clients via JMS is de-coupled as described above, it is still necessary for a client to know how to access the type of message or topic he is interested in. Thus, not the concrete provider, but the format or content type needs to be known in order to bind it.

4.1.3 Connecting JMS and SOAP-web services

To make use of JMS's advantages in a standard web service architecture, the following architecture combines both approaches.

From the client's perspective, the SOAP endpoint communicates with an endpoint in a standard manner. However, this endpoint isn't a standard web service but rather a protocol handler that listens for SOAP messages and passes them into a message queue. With the aid of a listener, the messages are then routed to the correct target (web service) asynchronously.

The motivation for this approach is that while the web services technology enables the execution of remote services, it does not provide a robust infrastructure for handling information. An enterprise-class application that communicates with web services however must ensure that the data can be handled appropriately. There are three reasons why standard web services can be enhanced by extending them as exemplified. Firstly, the data will be lost if the application fails because the data is not persisted, Second, if the system is inundated with orders, it must be able to handle the increased load. And finally, in the case that the application needs to communicate with a backend system, there must be a bridge for efficient and reliable communication. JMS supports these requirements and provides features to couple both systems, standard SOAP web service communication with the outer world and JMS messaging within a network to handle service requests.

Besides its significant advantages over web services, the JMS technology has some relevant drawbacks. They can be categorised as its potential overhead, additional complexity and the risk to form a communication bottleneck at the message server. The queues or topics need to maintain and control all incoming messages, perform (de-) serialisation of messages and schedule outgoing messages.

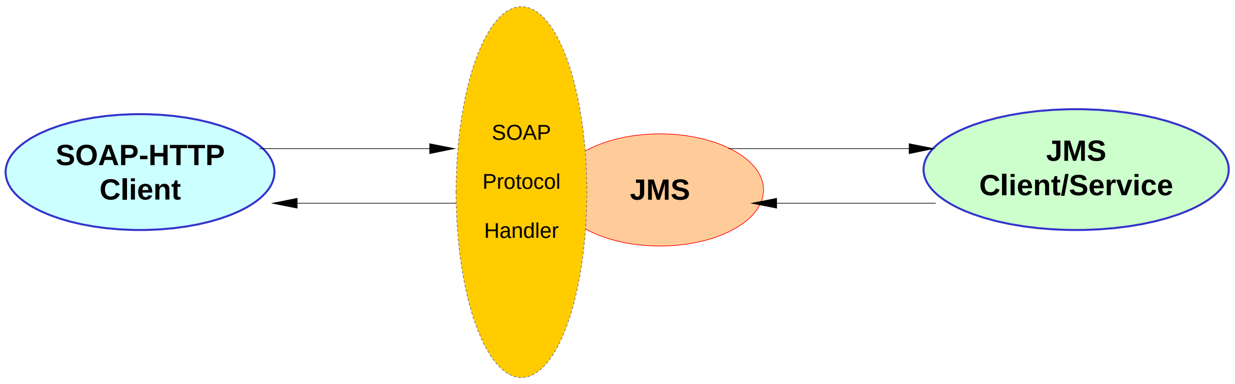

Figure 7: Architecture of connection between JMS and SOAP web services

4.2 JavaSpaces in SOC

In the literature JavaSpaces technology is said to be the „technology providing a high-level co-ordination tool for gluing processes together into a distributed application“[9]. With its concepts, it truly offers a loosely coupled communication with respect to location, time and reference. Its (not unlimited) scalability and fail-proof storage and exchange mechanism make it usable for services, which are reasonably executed asynchronously and in parallel.

As with JMS, JavaSpaces provide a communication middleware component for services to exchange data. Further, the processing directives are also encapsulated in the Entry objects. This makes architectures feasible where “workers” perform the computing by fetching any object from the space and doing its processing locally by calling the computing method.

In the JavaSpaces approach, the term service is interpreted in a slightly different way then it is the case for SOAP web services. The service as an open, self-descriptive component that provides a contract between provider and consumer in SOA does not explicitly exist in this architecture. JavaSpaces comprise object-centric data structures while each object can contain service logic. In fact, since the processing is performed by objects with access to the space, the service itself is distributed over an unknown and not controllable number of systems consuming space objects. When a service is distributed as a data structure written to the space, no service level agreement can be verified.

However, JavaSpaces can be combined with standard web services in the same manner as it was the case for JMS. The web service client hereby uses the well-known SOAP interface to call a gateway which distributes the service execution in a space. The difference hereby is that with JavaSpaces the main purpose is to distribute workload, not to call third-party systems (like JMS clients).

5 Conclusion and Outlook

JMS and JavaSpaces can provide significant components in a SOA by supporting loose coupling in the sense of location, time and reference. Especially when asynchronous communication between client and server is necessary, these concepts show their potential. Integration into standard web services can be realised by providing additional components in the architecture, which pass SOAP calls to an internal message server or space. But both concepts can also be integrated into a SOA without making use of web services.

The major limitations of both approaches are technical. They are restricted to the Java context and form certain bottlenecks in terms of memory and processing.

Both technologies provide an Java API for concepts that are significantly older. JMS 1.0.2b originates from 2001, the actual version 1.1 has been released in March 2002. The idea of message servers is however somewhat older. The concept of tuple spaces has been developed in 1982, while JavaSpaces has been released in 1998.

There has been a tremendous hype about these two technologies. JavaSpaces were claimed to be „[...] a full generation ahead of anything else on the market“ and that „the power of such systems would bring multicomputing to the masses.“[11]

Today, only few new articles or books are published about these two technologies. In the case of JavaSpaces, their application area seems to be restricted to scientific Grid-Computing while JMS is mainly used as SOAP transport mechanism.

Although both technologies do not yet play the role that have been predicted, the concepts behind both technologies bare high potential for a real loosely-coupled service-oriented computing environment. When certain issues in the architecture are addressed by applying additional mechanisms (e.g. components to administer and control JMS servers or the space workload), powerful SOAs can be set-up in the Java environment quite easily.

References

[1] S. Wilkes: SOA - Much More Than Web Services. *http://dev2dev.bea.com/pub/a/2004/05/soa_wilkes.html.*

[2] D. Krafzig, K. Banke and D. Slama. Enterprise SOA: service-Oriented Architecture Best Practices. *Prentice Hall*, 2004.

[3] Wikipedia: *Loosely Coupled, http://en.wikipedia.org/wiki/Loosely_Coupled.*

[4] B. Angerer and A. Erlacher: Loosely Coupled Communication and Coordination in Next- Generation Java Middleware, *today.java.net/pub/a/today/2005/06/03/loose.html*, 2005.

[5] J. Hanson: Take Advantage of the benefits of loosely coupled Web Services, *builder.com.com/5100-6386-1050425.html*, 2002.

[6] Java Message Service Specification: *http://java.sun.com/products/jms/.*

[7] G. van Huizen. JMS: An Infrastructure for XML-based Business-to-Business. Communication. *http://www.javaworld.com/javaworld/jw-02-2000/jw-02-jmsxml.html*, 2000.

[8] S. Maffeis. Introduction to the Java Message Service (JMS) Standard, *http://www.ch-open.ch/html/ws/ws01/11_jms.html#Download.*

[9] E. Freeman, S. Hupfer and K. Arnold: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley, 1999.

[10] JavaSpaces, *http://www.javaspaces.homestead.com/files/javaspaces.html.*

[11]R. Shah: The skinny on Jini, *http://www.javaworld.com/jw-08-1998/jw-08-jini.html*.

[12]Scientific Computing. Virtual Shared Memory and the Paradise System for Distributed Computing. Technical white paper, 1999.

[13]D. Gelernter: An Integrated Microcomputer Network for Experiments in Distributed Programming. PhD thesis, State University of New, 1982.

[14]Wikipedia: *Linda, http://en.wikipedia.org/wiki/Linda.*