Loosely Coupled Communication and Coordination in Next-Generation Java Middleware

by Bernhard Angerer and Andreas Erlacher
06/03/2005

Most companies own several independent systems with redundant data. These heterogeneous systems are a challenge to manage. A lot of effort is spent trying to reduce the number of systems and their complexity, as well as spare time, money and resources, by the consolidation and synchronization of the existing data. This article looks at the Java Messaging Service (JMS) and JavaSpaces for coupling autonomous systems efficiently. The following criteria have been taken into account: security, availability, reliability, maintainability, scalability, performance, caching, and notification of changes.

Contents

The Scenario
Benefits of Loose Coupling
Java Messaging Service (JMS)
JavaSpaces
Application Areas
JMS Examined
JMS Overview
Evaluating JMS
JavaSpaces
JavaSpaces Overview
Evaluating JavaSpaces
Epilogue/Acknowledgment
Resources

The Scenario

Distributed applications are notoriously difficult to design, build, and debug. The distributed environment introduces many complexities that aren't concerns when writing standalone applications. Issues like latency, synchronization, and partial failure need to be addressed. Asynchronous middleware offers loose coupling, which enables the use of architectures that cope with those issues in a better way (see [Goff03]).

Benefits of Loose Coupling

By being able to break the tight coupling inherent in many communication approaches, systems can be built that are more flexible, adaptable, and reliable. Location transparency, for example, allows components to exist anywhere, with the ability to migrate from machine to machine without changing how they communicate. In certain application semantics, time-decoupling is helpful, since the implementation of process flows becomes easier by eliminating the need to ensure that all speaking partners are up and running at the same time. Besides flexible communication, loose coupling facilitates the composition of large applications by letting the programmer easily add components without redesigning the entire application. It also facilitates the binding of additional resources that can act in a hot standby mode in order to reach disaster reliability or to distribute the load when the system needs to handle high volumes.

Another technology used to exchange data is web services. Most of the existing implementations are "document-centric" communications and are implemented by extending the remote procedure call (RPC) model. The underlying patterns and models are changing these days because of the adoption of web services. Although the web service model supports asynchronous communication, web services operate with a coarser granularity (see [Dim05]) and offer loose coupling in different way than what JMS or JavaSpaces are doing. Communication participants can be seen as autonomous satellites that maintain their states themselves and can be bound by contracts. There is no instance that can be used for overall synchronization and persistency. Therefore, they are not discussed here.

Java Messaging Service (JMS)

JMS offers loosely coupled communication through messages that are delivered by a message server. The communication and data exchange are asynchronous and decoupled. The sender does not need to be responsible for the delivery procedure or issues such as:

- Is the receiver up and running?
- Could the receiver consume the message?
- What happens in case of failure during delivery?

The communication channels are queues. Any client can participate in the communication by registering with a specific queue. There are existing implementations without a central message provider. The JMS specification (see [Sun02]) recommends the usage of a JMS provider, that is why the "only client" scenario is not handled in this article.

JavaSpaces

JavaSpaces offer loosely coupled communication with the paradigm of distributed shared memory (called the "space"). By writing objects (data structures such as trees or lists) into local memory (without direct network calls), the application hands over to the space all of the bothersome issues of distribution. The objects are also replicated to other machines that showed interest in these objects. In this way, a shared, in-memory communication- and coordination-context system is created that offers stateful transaction processing and represents a common view entity for all participants. JavaSpace communication is decoupled in terms of time, space, and reference (see [Ang02]). The only shared reference is created through logical namespaces that define so-called distributed data structures (see [Free99]).

Application Areas

JMS is part of the Java 2 Enterprise Edition (J2EE) specification. It is used:

- Within J2EE servers for asynchronous communication
- In heterogeneous environments with decoupled systems to communicate over system boundaries.

JMS is always appropriate within systems where the information provider should not be aware of any information consumer. Depending on the model used (publish/subscribe or point-to-point), the consumer also does not necessarily know the producer. As described later, there are several limitations regarding the acknowledgment of information receipt. JMS is available as a separate component that can be easily integrated into systems based on the Java 2 Standard Edition (J2SE).

Today, JavaSpaces is used by the industry in many different scenarios. Two clear motivations are:

- Performance enhancement
- Complex real-time integration (grid computing)

The combination of distributed caching and easy distribution of load (see [Li03]) makes the JavaSpaces platform an ideal one for performance enhancements or bottleneck eliminations. JavaSpaces exploit their full potential when configured as a distributed space, which means that the engine consists of a federation of kernels--a peer-to-peer (P2P) approach (see [Gig04]). (In writing the specification, the Sun team has tried hard to enable the development of a replicated space service implementation as well as a standalone one. This article is going to focus on the former case). This means that the load of the engine itself is also distributed and server bottlenecks are prevented. The massive scalability enabled by the P2P architecture in combination with the introduction of a distributed, stateful communication and coordination resource is the primary motivator for its usage in complex integration problems. The processing of stateful transactions with additional meta-information at hand makes it an ideal tool to integrate many logical communication paths with different transactional and non-transactional resources in a real-time manner. This is why spaces are often used within grid computing.

JMS Examined

JMS offers an infrastructure for several systems to share data (see [Hae01]). In general, messages are produced by one or several clients, delivered by a message provider (server), and then consumed by one or several clients. Figures 1 and 2 show the architecture of a heterogeneous system using JMS, as well as a look at how communication flows within JMS.

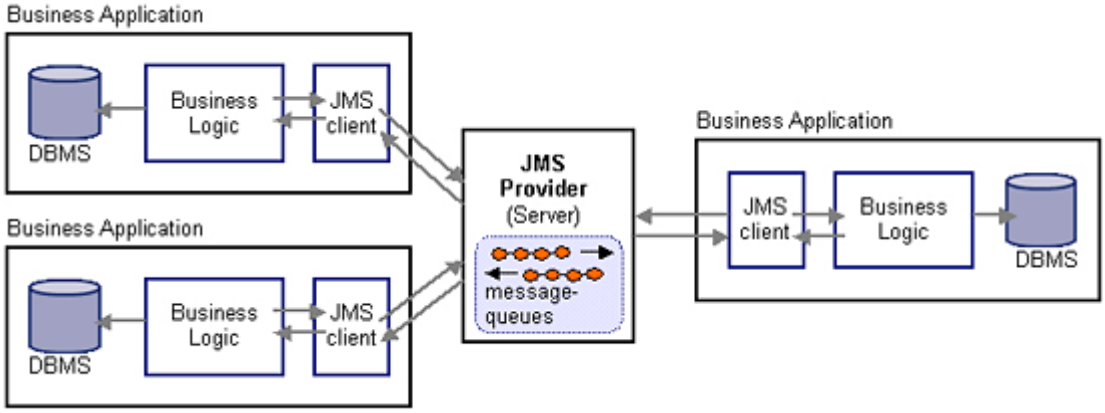

Figure 1. Heterogeneous system with JMS

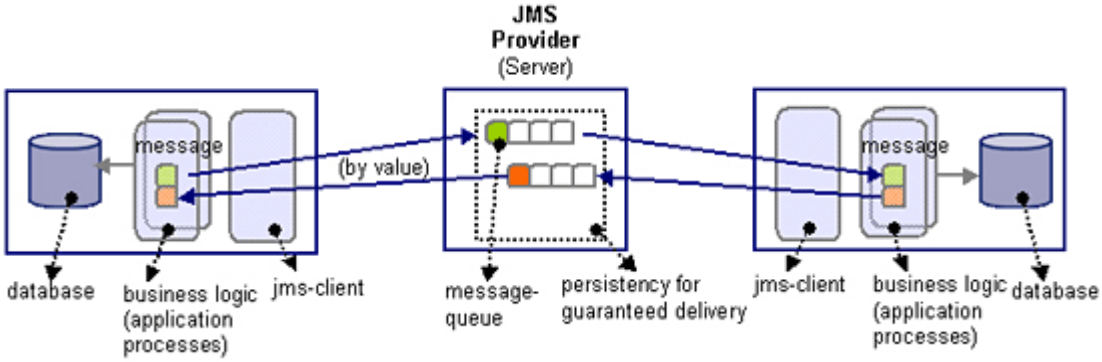

Figure 2. Communication within JMS

JMS Overview

This section describes the components of a JMS-based system.

JMS provider: Most JMS providers support different delivery modes. The OpenJMS server used for this study provides different mechanisms to connect to the server, such as TCP, TCPS, HTTP, HTTPS, and RMI. HTTP(S) may be necessary if firewalls, etc. prevent the usage of TCP, TCPS, or RMI. Additionally, an "embedded" mode connects to an embedded OpenJMS server; i.e., a server running in the same JVM as the clients. This avoids the overhead of network serialization.

Mechanisms for communication: With JMS, two different mechanisms for communication can be used by the clients:

- Point to point (P2P): The client directly connects to another client using a specific queue registered with the JMS provider. The client can send and/or receive messages from/to that queue.
- Publish/subscribe (Pub/Sub): The client subscribes to a topic he is interested in. Any message related to the topic published by a publisher is received by the client.

Messages are delivered using sessions. A session is a single-threaded context for producing and consuming messages and is provided by the JMS provider's connection. It defines a serial order for the messages it consumes and the messages it produces. Depending on the messages to share, a participant in the system has to be a JMS message producer and/or a JMS message consumer.

Messages: Message objects are composed of three parts. The header contains the information needed by client and provider to identify and route messages. With properties, information can be added to the header. These can be standard (optional header fields defined by the JMS specification), or application- or provider-specific fields. The body is the information to be delivered.

JMS defines five different types of messages bodies (`StreamMessage`, `MapMessage`, `TextMessage`, `ObjectMessage`, and `BytesMessage`), which differ in their content. The lifetime of a message can be defined using its expiration timestamp. The provider uses the timeframe to determine how long attempts have to be performed to send the message. Improper setting of the timestamp using the Pub/Sub mechanism can lead to non-delivered messages to reasonable inactive durable clients when the inactivity time exceeds the timestamp. Any way it can happen that messages are not deliverable and with respect to reliability, it can be useful to set the lifetime timestamp to control message delivery (i.e., in a handshake scenario).

Message Selection: JMS provides the possibility for clients to define messages they are interested in by specifying selection criteria in the message header and properties. Message selectors cannot reference message body values. The filtering is done by the message provider, and the criteria are specified by the client using a string whose syntax is based on a subset of the SQL92* conditional expression syntax. A message selector matches a message if the selector evaluates to true when the message's header field and property values are substituted for their corresponding identifiers in the selector.

Message Delivery: JMS supports two modes of message delivery:

- The `NON_PERSISTENT` mode is the lowest-overhead delivery mode because it does not require that the message be logged to stable storage. A JMS provider failure can cause a message to be lost.
- The `PERSISTENT` mode instructs the JMS provider to take extra care to ensure the message is not lost in transit due to a JMS provider failure.

A JMS provider must deliver a non-persistent message at most once. This means that it may lose the message, but it must not deliver it twice. A JMS provider must deliver a persistent message once and only once. This means a JMS provider failure must not cause it to be lost, and it must not deliver it twice. Redelivered messages must be labeled by the JMS provider. This is done by setting a header field. Redelivery can occur if the client doesn't acknowledge a message for a reasonable time. The client has to decide what action to perform if it indicates that the message was redelivered. For critical messages, it is useful to cache all received messages by the client. JMS defines that messages sent by a session to a destination must be received in the order in which they were sent.

Message Consumption Synchronous and asynchronous modes are available for the P2P and the Sub/Pub messaging model:

With the synchronous mode, the client calls the method `receive()` of any `MessageConsumer` instance (i.e., `QueueReceiver` for P2P and `TopicSubscriber` for Pub/Sub) and is blocked until a message is received. There are several variations of `receive()` that allow a client to poll or wait for the next message. For example, one implementation of the `receive()` method accepts a parameter that defines a timeout to wait for the next message.

With the asynchronous mode, the client must implement the `javax.jms.MessageListener` interface and overwrite the method `onMessage()`. In P2P mode, the client has to be registered with the `QueueReceiver` instance of the `QueueSession` instance of the receiving queue. In Pub/Sub mode, the client has to be registered with all `Topic`s to which it wants to subscribe. This is done by registering it to all instances of `TopicSubscriber`. By definition, a subscription to a topic is not durable. The subscriber receives only messages while it is active. If a client needs to receive

all messages dedicated to a topic, it has to be durable. The durable subscribers are maintained in a special record by the JMS provider. The provider ensures that all messages from the topic's publishers are retained until they are acknowledged by this durable subscriber or they have expired.

Message Acknowledgment: With asynchronous delivery, the feature of *acknowledgment* can influence reliability. The client and the server establish a contract for acknowledgment of delivered messages. JMS supports the types `AUTO` (automatic acknowledgment by the session), `CLIENT` (manual acknowledgment by calling a method of the message object), and `DUPS_OK` (the session lazily acknowledges the delivery of messages). With transacted sessions, any acknowledge mode is ignored. The delivery is controlled by the session's commit and rollback mechanisms. Sending transactional grouped messages followed by asynchronous receives within a transaction should be avoided due to possible long delays between send and receive depending on the processes involved.

Transactions: A session can be marked as `TRANSACTED` to support a single series of transactions that combine work spanning its producers and consumers into atomic units. Each transaction groups a set of produced messages and a set of consumed messages into an atomic unit of work. Any acknowledge mode is ignored with transacted sessions. Messages delivered to the JMS provider in a transaction are not forwarded to the consumers until the producer commits the transaction. It depends on the JMS provider how uncommitted or not rolled back transactions are handled when a client restarts. This is a critical topic, because it can lead to unpredicted behavior if this scenario is not handled with care. Distributed transactions are not a feature of JMS. A JMS client may participate in distributed transactions by implementing the Java Transaction API (JTA). The transactions must be handled by the environment the client is running in; i.e., a JMS provider can optionally support distributed transactions via JTA (see [Sun02]). JMS implementations can form their own distributed transactions through the Java Transaction Service (JTS) to combine message activities with database updates and other JTS-aware services. When a JMS client is run from within an application server (such as an Enterprise JavaBeans Server), distributed transactions should be handled automatically.

Evaluating JMS

In this section, we look at JMS in the context of a collection of criteria that is critical in distributed applications.

Security: JMS does not specify an API for controlling the privacy and integrity of messages. It also does not specify how digital signatures or keys are distributed to clients. Security is considered to be a JMS-provider-specific feature that is configured by an administrator rather than controlled via the JMS API by clients. Clients will get the proper security configuration as part of the administered objects they use (see [Sun02]). Authentication is supported by JMS connections. When creating a connection, a client may specify its credentials as name/password. If no credentials are specified, the current thread's credentials are used.

Availability: Availability is mostly determined by the JMS server. As destinations and connections are created by JNDI lookup, the availability of the server can be enhanced by proxying. This can lead to problems with reliability.

Reliability: The Quality of Service (QoS) of the whole system is determined by the reliability of the JMS provider. JMS messaging provides guaranteed delivery via the once-and-only-once delivery semantics of persistent messages, even if the JMS provider shuts down and has to be restarted. In addition, message consumers can ensure reliable processing of messages by using either client-acknowledge mode or transacted sessions. JMS does not define system messages such as delivery notifications. If an application requires acknowledgment of message receipt, it can define an application-level acknowledgment message object.

Scalability: Depending on the JMS provider, the system can be scaled by clustering. If additional instances of JMS providers are installed, the clients must be accordingly configured--at least, they must know the IP address of the new instance.

Performance: The JMS server can lead to performance problems, as it can be a bottleneck. Using the non-persistent mode with message delivery can enhance performance at the cost of reliability.

Caching: Caching of information delivered by messages is up to the clients. In the Pub/Sub-model, durable subscriptions can be used as a cache provided by the JMS server (see Message Consumption).

Notification of Changes: In general, JMS messages are passive information containers. Delivered messages cannot be altered by the sender after sending them to the destination. The Publish/Subscribe mechanism can be used as a notification mechanism to implement the Observer design pattern. Additional clients can use the JMS `MessageListener`/`MessageConsumer` model to get notifications from a message producer by registering an object that implements the JMS `MessageListener` interface with a `MessageConsumer`. As messages arrive for the consumer, the provider delivers them by calling the listener's `onMessage()` method.

JavaSpaces

JavaSpaces offers a distributed infrastructure that creates a distributed virtual shared memory, which functions as a communication and coordination pool. In general, distributed data structures are created, by which one or several "clients" or participants communicate and coordinate themselves. A "server" physically doesn't existing. It is implicitly created through an active replication protocol that unites all participants or nodes that take place in the communication. Figures 3 and 4 show the architecture of a JavaSpaces heterogeneous system as well as the details of communication within such

an application.

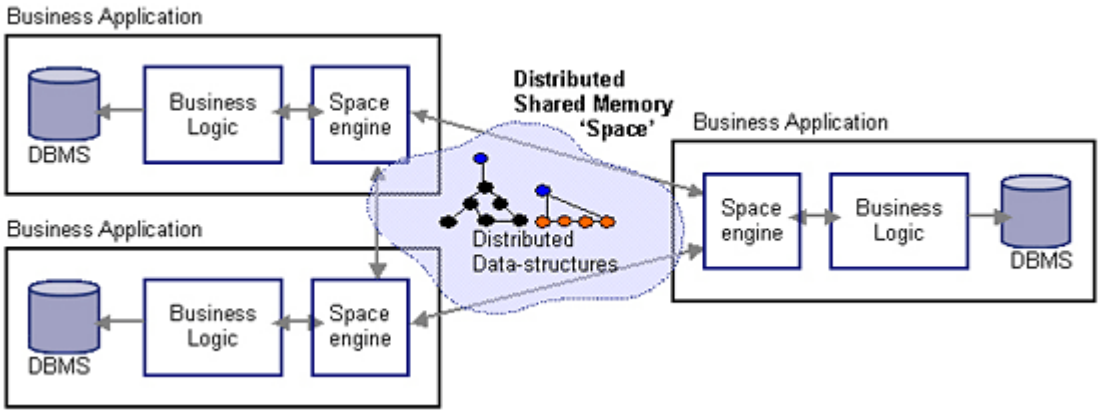

Figure 3. Heterogeneous system with JavaSpaces

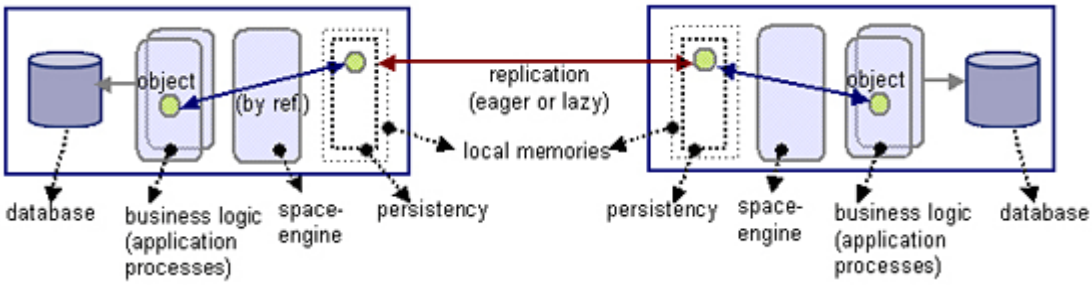

Figure 4. Communication and coordination within JavaSpaces

JavaSpaces Overview

The mechanism for JavaSpaces is quite different from that for JMS. This section provides you with a look at how JavaSpaces works.

JavaSpaces "virtual server": The 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 (pull or push update policies may be implemented, depending on the vendor implementation) is incorporated into the space infrastructure. RMI (or another protocol, depending on the vendor) is 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.

Mechanisms for communication: With JavaSpaces, any type of communication design pattern can be used by the participants. In particular, the level of coding effort stays low, regardless of how complex the communication patterns are, because every communication is implicitly bi-directional (see Figure 5 below). The following five patterns are captured in order to make things clearer:

- Point-to-point (P2P): A peer writes an entry into space. Another peer has a notification registered or does a blocking read on that entry. After the communication, the entry can be left inside the space (caching) or removed (consuming read through a take operation of the receiver).
- Publish/subscribe: A peer writes an entry into space. Other peers have notifications running on that entry. The entry objects in space contains additional information to determine the overall status of the communication to make clear when entries can be removed, and so on.
- Master/worker: Basically the same as publish/subscribe with the difference that results of the subscriber or worker are collected again in space to be passed back to an initiating publisher or master.
- Client/server: Basically the same as master/worker, but this time the communication paths work the other way around. The server or master never starts a communication, but is always contacted by the clients or workers.
- Agent-based: Arbitrary or complex communication patterns that are following application semantics that represent various views and logical communication paths and levels. This is handled in a straightforward way because the space objects are not only transporting communication events and data but also function as a "speaking context" between the peers.

The following picture illustrates the differences between communication patterns of the two architectures. JMS defines point-to-point and publish/subscribe patterns, but more complex message exchanges are certainly implemented, too. These complex scenarios show one of the main differences between JMS and spaces. Internally, the space API works via serialization when entries in space are accessed. However, the architecture and the paradigm of space-based data structures imply a "reference-based" access procedure (versus the "by value" procedure of JMS).

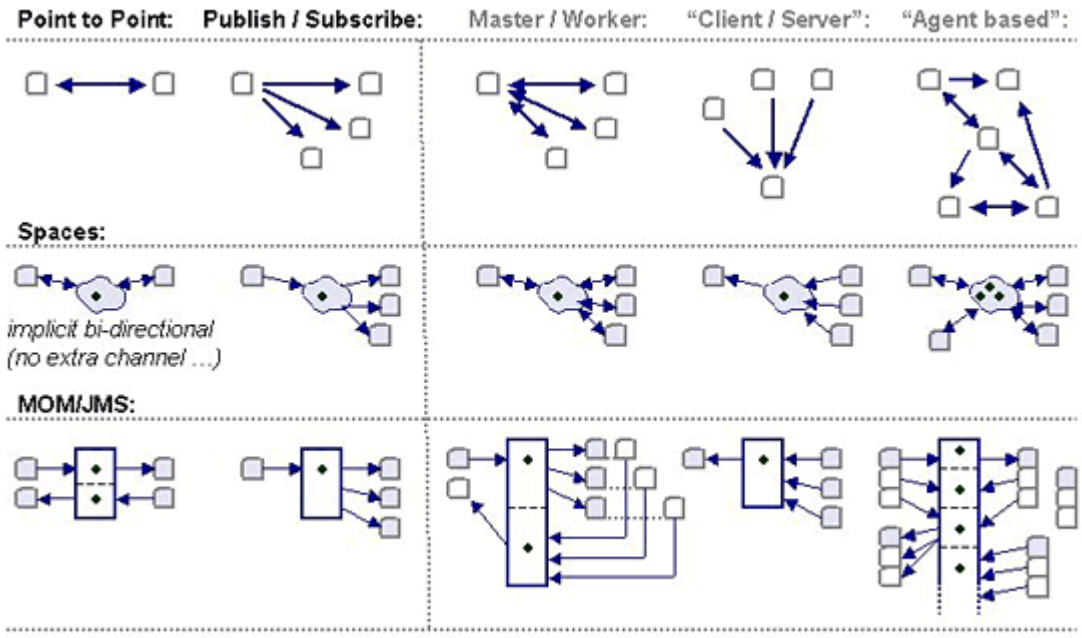

Figure 5. Communications patterns

With JavaSpaces developers do not follow the "message deliver" or "messaging" paradigm. 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 that creates the illusion of a single address space where the code looks like it's running on a single machine.

Space objects: Space objects have to implement the entry interface. Read operations can grab those objects following template matches:

1. The template's type is the same as the entry's, or is a supertype of the entry.
2. Every field in the template matches its corresponding field in the entry (either exact value matching or field values are arbitrary when they are assigned with a wildcard, respectively a null-value).

Every object is equipped with an expiration timestamp called the *lease* (`Lease` interface). So it is guaranteed that space resources are released again in the case of partial failure of resource holders. The holder of the lease may renew or cancel the lease before it expires. In this way, time-critical operations can be kept on a small granularity for overall failure detections.

The space API itself is very minimalist (basically a four-method API: read, write, take, and notify) and doesn't imply assumptions about application semantics. One common question that arises within this context is how to make sure that the receiver gets entries in the right order, because the space doesn't provide any functionality for ordering or sorting. Questions of this type are solved with the possibilities that are within the object-oriented approach itself. Index properties are simply used to number all entries when they are written into the space. Thus, the receiver refers to this number or index and can access particular entries, or a whole list in the proper order.

Space object delivery: JavaSpaces supports two modes:

- `TRANSIENT` or `NON_PERSISTENT`: This defines an in-memory space that hold all of its data in distributed memory, which results in fast access. However, memory spaces are bounded by the amount of distributed memory in the system, and are vulnerable to server crashes.
- `PERSISTENT`: A persistent space uses a DBMS back end to persist its data in a transparent manner, while still caching some of the data in memory. (Server implementations use a "buffered persistency" through log files to speed up performance). Persistent spaces do not lose data as a result of server reboots/crashes and can hold a large amount of data.

JavaSpaces realize guaranteed delivery through the combination of replication and distributed persistency. If a node is down and doesn't receive update information, there is no instance that buffers those activities and repeats update requests. All space operations are inherently fail-safe (and may also provide fail-over capabilities, depending on the vendor implementation).

Acknowledgment/consumption: The space approach doesn't provide a mechanism for acknowledging procedures and the like. Peers simply read, write, and take entries from the space. Because the space itself is a distributed resource, it is

the ideal tool to implement system-wide monitoring. On a process level, space objects can function as live beats and tell participants if a peer is down or the network is broken.

Transactions: Using transactions in space-based operations is fairly straightforward. A process obtains a transaction from a manager and then passes that transaction to each space operation (which may occur over one or more spaces). If the commit is successful, then all operations under the transaction are guaranteed to have completed successfully. If the transaction is aborted, it's as if none of the operations occurred, leaving the space unchanged. It is important to note that transactions behave with lease times, like all of the other space operations do. If a transaction's lease expires, the transaction is automatically aborted, and none of the operations performed within the transaction will be reflected. In this way, the space can be kept stable and consistent when processes have unpredicted errors and break away during changes (take and write operations). During long-term transactions, the lease must be set to a smaller value than the duration of the transaction and has to be renewed (`LeaseManager`) during execution. This defines the granularity in which the space is aware about the status of its peers.

Evaluating JavaSpaces

In this section, we evaluate JavaSpaces against the same set of criteria used to measure JMS.

Security: The JavaSpaces API does not address security. Comprehensive security issues were addressed with the release of Jini 2.0 (see [Jin04]). This article is concentrating on the level of JavaSpaces and therefore this issue is out of scope. However, the major vendors are offering pluggable login and authorization features as well as transport encryptions.

Availability: The availability issues are portrayed "differently" in a space system. The following two points need to be considered:

- Peer-to-Peer (P2P) or federated architecture
- Communication=replication

Because the space is a distributed shared memory created by a federation of engines, there is no central server (single point of failure). A peer that takes part in the communication gets a local copy of an entry on its machine (replication). The fusion of the communication and replication models enables transparent fail-over behaviors with additional hardware that works in a hot-standby mode in the case of failure.

Reliability: Overall system failure semantics or fail-over capabilities through redundant, hot-standby resources can be easily implemented using the space as a live-beat dashboard. Additional hardware resources in combination with the space-inherent data replication can lead to disaster reliability in these scenarios.

Once-and-only-once delivery semantics don't match the space approach, because of its stateful information-sharing architecture. The space functions as a communication and storage mechanism, in which non-consuming reads are very usual. Special semantics always need to be bundled within the distributed data structures, respectively the properties of the entries itself. This way additional information for constraints, up-to-date-ness, or type of the information processed is attached.

Scalability: The two main points describing the architectural approach (the P2P engine and the fusion of communication and replication) that enable high availability are also responsible for very straightforward characteristics when it comes to scalability issues. This can be described on a physical as well as on a logical level:

- Inherent replication and distributed caching (physical level)
- Logical "communication-space" (logical level)

Here, the key point is that these two aspects play hand in hand. The communication space enables high utilization of the distributed environment. If an application is confronted with high load, a master-worker pattern can distribute worker processes across many machines. The distributed memory takes care that all workers have the same view on data, and session or context information. This is done through application-aware replication (only entries that are needed will be replicated) that automatically functions as a distributed cache. All communication and coordination is processed through the entries in space (a master writes job requests into space and workers are waiting for them) which means that up- or down-scaling is totally transparent for the application code and can be accomplished even during runtime. It is important to note that the space enables massive scalability, because there is no central space server that needs to process all of the replication and synchronization work. In fact, the load of the space engine itself is also distributed across nodes as well as the application code.

Performance: The correct use of JavaSpaces (not using it as a database replacement, for example) leads to performance enhancements rather than creating performance problems or bottlenecks. However, performance is mostly determined by the space server implementation (distributed hash tables are the main instrument in order to establish fast concurrent access and the processing of optimistic locking protocols).

Caching/notification of changes: Since the space is based on replication, distributed caching is a core part of the architecture. If an entry once resides on a machine, an update somewhere else in the space causes synchronization and

an update of each of the replicated copy's respective entries. The tradeoff between network traffic and performance is controlled via the configuration of several logical spaces (segmentation) as well as push (active) or pull (passive) update policies.

The JavaSpace interface provides a notify method that allows processes to register an object's interest in the arrival of entries that match a specific template (notify/`RemoteEventListener`).

Conclusion

The intent of this article is not to compare JMS and JavaSpaces, but rather to give an understanding about the positioning and differences between these two architectures. It can be asserted that JMS is designed for information delivery, whereas JavaSpaces can be called an information-sharing infrastructure. The latter is used in a correct way when it holds process- and meta-information of an application, which implicates message delivery as well as storage. This mixture of aspects covered by a space is one reason why it's misunderstood in many cases versus JMS, where the main purpose was rather coherent from the beginning.

Epilogue/Acknowledgment

There are many different approaches and levels of how to deal with these topics. It was our intention to write this article more from a messaging standpoint, which primarily may strike the "space guys" and explains why certain things that are typical for either of the techniques are not discussed (e.g., the code-moving feature of spaces). However, we would like to thank the following experts from the field for their cooperation and help: Max Goff, Dan Creswell, Gerald Loeffler, Dave Sag, Nati Shalom, and Ronald V. Simmons.

Resources

- [Goff03] Max K. Goff, Network Distributed Computing: Fitscapes and Fallacies, Prentice Hall, 2003, ISBN 0-13-100152-3
- [Dim05] Labro Dimitriou, "Distributed Parallel Computing with Web Services," WebServices Journal, 2005
- [Sun02] Sun Developer Network, Specification of JMS 1.1, 2002
- [Ang02] Bernhard Angerer, ONJava.com: "Space-Based Programming," O'Reilly, 2002
- [Free99] Eric Freeman, Susanne Hupfer, and Ken Arnold, JavaSpaces Principles, Patterns, and Practice, Addison-Wesley, 1999, ISBN: 0-201-30955-6
- [Li03] Sing Li, "High-Impact Web Tier Clustering, Part 2: Building Adaptive, Scalable Solutions with JavaSpaces," IBM developerWorks, 2003
- [Gig04] GigaSpaces Technologies Ltd., "GigaSpaces Peer-to-Peer Cluster Patterns," 2004
- [Hae01] Monson-Haefel and Chappell, Java Message Service, First Edition, O'Reilly, January 2001, ISBN: 0-596-00068-5
- [Jin04] Jini.org: "Getting Started with Jini 2.0," 2004

Bernhard Angerer is a dedicated software engineer.

Andreas Erlacher is a Software Architect and certified Process Manager.