Using Space-Based Programming
For Loosely Coupled Distributed Systems

Abstract

One of the problems of highly distributed systems is figuring out how systems discover each other. After all, the whole point of having distributed systems is to allow flexible and perhaps even dynamic configurations to maximize system performance and availability. So how do these distributed components of one system or multiple systems discover each other? Furthermore, once these systems are discovered how do we allow enough flexibility, such as rediscovery, so as to allow fail-safe operation of such systems? Space based programming may provide us with a very good answer to these questions and more.

This article describes what a space is and how it may be used towards mitigating some of the issues mentioned above. It then discusses a technique of converting an ordinary message queue into a space. A list of resources is provided at the end of the article for readers interested in learning more about space-based programming and applications.

What is a Space?

Conventional distributed tools rely on passing messages between processes (asynchronous communication) or invoking methods on remote objects (synchronous communication). A space is an extension of the asynchronous communication model in which two processes are not passing messages to one another. In fact the processes are totally unaware of each other.

Let's look at figure 1 for a moment. Process 1 places a message into the space. Process 2, which has been waiting for this type of message, takes the message out the space. Process 2 processes the message and based on the results places another message into the space. Process 3, which has been waiting for this type of message, takes this message out of the space.

Following are the highlights of the preceding discussion:

1. The space may contain different types of messages. In fact I used the term "message" for clarity. These messages are actually just "things", i.e. the message may be an object, an XML document, or anything else that the space allows to be put in it. In figure 1 the different shapes in the space illustrate the different types of messages.
2. The three processes involved have no knowledge of one another. All they know is that they put a message in a space and get a message out of the space.
3. As in the message passing scenario, we are not limited to two processes communicating asynchronously, but rather any number of processes may communicate via a common space. This allows the creation of extremely loosely coupled systems that can be highly distributed, extremely flexible, and can provide high availability and dynamic load balancing.

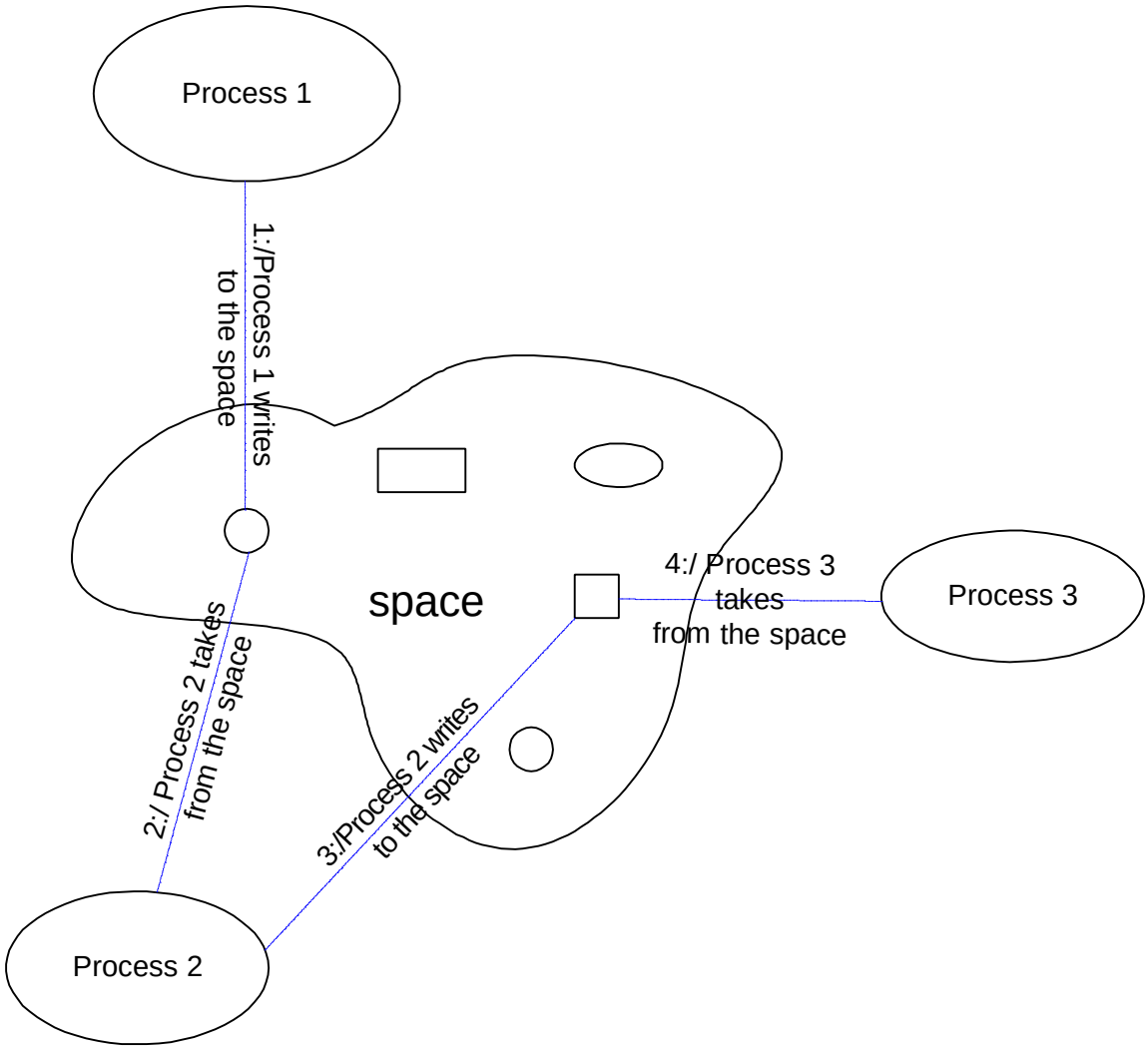

Figure 1

Let's look at a more specific example this time. A common encryption method is the use of "one-way" functions, which take an input and like any other function generate an output. The distinguishing feature of such functions is that it is extremely difficult to compute the input that was given to the function to get the output, i.e. to compute the inverse of the function. Hence, the term "one-way" function. So, instead of trying to figure out the inverse of the function to get the input required for the given output, an easier way may be to take all possible inputs and compute the output for each input. When we get an output that matches the one we have, we have found the right "input". But this can be extremely time consuming gi ven the vast number of possible inputs. Let's assume that passwords cannot be more than four characters in length and only

alphanumeric ASCII characters[1] are used. This gives us 14776336 possible passwords[2]. Furthermore, let's use the "brute" force technique to break the password. Assume that the main program breaks the input set into 16 pieces and puts each piece along with the encrypted password in the space. The password-breaking programs are watching the space for such pieces and each available program immediately grabs a piece and starts working. The programs continue till there are no more such pieces available or until the password has been broken. If the password is broken, the breaking program puts the solution in the space, which is picked up by the main program. The main program then proceeds to pick up all the remaining pieces, since it has already found the solution it needs. The main program never knew how many password-breaking programs were available nor did it know where these were located. The password-breaking programs had no knowledge about one another or about the main program. If there were 16 password-breaking programs available and each one was on a separate machine, we would have had 16 machines working on breaking the password simultaneously! Also, to add new password-breaking programs, no change to any configuration of the system is required. This is why spaces are so good for fault-tolerance, load balancing, and scalability.

As seen above, spaces provide an extremely powerful concept/mechanism to decouple cooperating or dependent systems. The concept of a space is not new. "Tuple spaces" were first described in 1982, in context of a programming language called "Linda". Linda consisted of "tuples", which were collections of data grouped together, and the "tuple space", which was the "shared blackboard" from which applications could place and retrieve tuples. However, the concept never gained much popularity outside of academia. Today, spaces may be an elegant solution to many of the traditional distributed computing dilemmas. In fact in recognition of this fact, JavaSoft has created its own implementation of the space concept called "JavaSpaces" and IBM has created "T Spaces", which is much more functional and complex than JavaSoft's JavaSpaces. We will not be discussing IBM's T Spaces in this article.

We are now in a position to describe some of the key characteristics of a space:

1. Spaces provide shared access.
A space provides a network-accessible "shared memory" that can be accessed by many shared remote/local processes concurrently. The space handles all issues regarding concurrent access, allowing the processes to focus on their task at hand. At the very least spaces provide processes with the ability to place and retrieve "things". Some spaces also provide the ability to read/peek "things", i.e. to get the "thing" without actually removing it from the space, thus allowing other processes to access it as well.

2. Spaces are persistent.
A space provides reliable storage for processes to place "things". These "things" may outlive the processes that created them. This allows the dependent/cooperating processes to work together even when they have non-overlapping lifecycles. This boosts the fault-tolerance and high availability capability of distributed systems.

3. Spaces are associative.
Associative lookup provides processes a way to "find" the "things" that they are interested in. Since many processes may be using/sharing the same space, there may be many different types of "things" in the space. It is important that processes are able to get the things that they require without needing to filter out all the "noise" themselves. Spaces allow this by allowing processes to define filters/templates that instruct/direct the space to "find" the right "things" for that process.

These are just a few key characteristics of spaces. Many commercial space implementations, such as the ones from JavaSoft and IBM, have additional characteristics such as the ability to perform "transacted" operations on the space.

[1] Hence the character set is [0…9, a…z, A…Z]
[2] Number of passwords = $(62)^4$ = 14776336

JavaSoft's Implementation: JavaSpaces

JavaSpaces technology is a new realization of the "tuple spaces" concept described above. It is an implementation that is available freely from JavaSoft. JavaSpaces is built on top of another complex technology called Jini. In a nutshell Jini is a Java based technology that allows any device to become network aware. Jini provides a complex yet elegant programming model that realizes the Jini team's vision of "Network anything, anytime, anywhere". The goal of JavaSpaces is to provide what might be thought of as a "file system for objects". Like every other JavaSoft API, JavaSpaces provides a simple yet powerful set of features to developers. However, there are some drawbacks to JavaSpaces as I see it. First of all, the implementation of JavaSpaces available from JavaSoft is fairly complex to install to say the least. Secondly, the fact that it builds on top of Jini makes it a little too heavy, especially if there are no plans to use Jini elsewhere in the project. Thirdly, JavaSpaces relies on Java RMI. The suitability of Java RMI for highly scalable commercial applications is a topic of debate among many software gurus. Fourthly and finally, JavaSpaces only works with serializable Java objects.

Creating your own space implementation

As discussed above, there are commercial implementations of spaces available in the market. However, there are several reasons for creating your own. If you work in a start-up company, budget constraints may be a big reason. Also, the functionality offered by a commercial implementation may just be too much for the job at hand. Not only may this result in a larger learning curve, it may even slow your application down due to the sheer size of the memory footprint. Finally, it's always fun creating your own implementation[3]☺.

At Online Insight, we decided to create our own implementation. The primary reasons for our decision were our limited set of requirements and the extremely lightweight implementation [4] that we required for achieving our scalability and performance goals.

Our requirements can be summarized as follows:

- The space must support shared access.
- The space must be persistent.
- The space must provide the ability to specify a filtering template.
- The space must allow one "thing" in the space to be accessed by only one process/application at a time i.e. we do not support the "read" operation.
- The space must perform and scale well under load[5].
- The space must be accessible to other CORBA objects.
- The space must be not impose a limitation to what you can put in it [6].
- The space must not impose size limitations on what you put in it [7].

Note that the first three requirements are also key characteristics of a space.

---

[3] Take this with a grain of salt. This comment is not meant to stir up the whole Buy Vs Build debate.
[4] Even though IBM T Spaces are lightweight, our implementation is even lighter at the cost of limiting some functionality that we do not require anyway.
[5] For requirements fanatics, this may be a little bit too vague.
[6] Unlike JavaSpaces for example.
[7] Note however that the underlying hardware, e.g. Disk space, available memory, etc. may impose a limitation.

At the same time, we were evaluating message queue type software, more specifically, Java Message Service (JMS) implementations, when we realized that we could build our space facility on top of one of these queues.

JMS is an API for accessing enterprise-messaging systems from Java programs. JMS defines a common set of enterprise messaging concepts and facilities. It attempts to minimize the set of concepts a Java language programmer must learn to use enterprise-messaging products, such as IBM MQSeries. It strives to maximize the portability of messaging applications. JMS does not however address load balancing/fault tolerance, error notification, administration of the message queue, or security issues. These are all message queue vendor specific and outside the domain of the JMS.

By using message queues that expose a JMS interface, we allow ourselves the flexibility to switch vendors of message queues in case we discover that they do not meet our scalability requirements. This separation of implementation from interface is an important design pattern [8]. Since each JMS implementation has its own unique way of getting the initial connection factory, we defined a Java interface with one method "getConnectionFactory" that returns the initial connection factory. Each space is configured through a properties file. One property in this properties file is the fully qualified [9] name of the class that implements this interface. There is one such class for each JMS implementation supported by the space. For example, we created one class for Sun's Java Message Queue, and one for Progress Software's SonicMQ. By doing this, changing the underlying message queue used by the space is simply a matter of changing the name of the Java class in the properties file for the space. Therefore, if one vendor's message queue does not live up to our expectations we can quickly switch to another one.

The space implementation itself is a CORBA object that has the following interface

```
interface Space
{
      void write(in ByteStream blob) raises (SpaceException);
      ByteStream take() raises (SpaceException);
      void write_filter(in ByteStream blob, in FilterSeq f)
                  raises (SpaceException);
      ByteStream take_filter(in FilterSeq f) raises (SpaceException);
      ByteStream take_filter_as_string(in string f)
                  raises (SpaceException);

      void shutdown();
};
```

The type ByteStream simply evaluates to a stream of bytes. Hence anything that can be represented as a stream of bytes, such as a CORBA object IOR, a serialized Java object, an XML document, etc. can be stored in the space and retrieved.

Each space instance has three properties: a name, a property that indicates if this instance of the space is persistent, and a property that indicates if this instance of the space allows filters. The reason there are properties to turn the persistence and filtering off is purely for performance. Not all spaces in our application domain are required to be persistent, in which case persistence is a performance bottleneck because it involves writing out to a database or similar storage mechanism. Similarly, if filtering is not required it is a performance bottleneck. As mentioned above, each space is configured through a properties file. This properties file has the property

[8] See the Bridge design pattern in Design Patterns, Elements of Reusable Object-Oriented Software, Gamma et al.

[9] name of the class with the package names included in dot notation

indicating the space name, the persistence status (on/off), and the filtering status (on/off) of the space.

An example of the properties file used in configuring the space is shown below:

```
SpaceName=MySpace
AllowFilter=true
Persistent=true

# The factory to use to get the initial Connection Factory
SpaceFactory=SonicMQSpaceFactoryImpl
```

The “SpaceName” property is the name of the space, “AllowFilter” is a boolean property where true means the space turns filter support on, and “Persistent” is a boolean property where true means the space turns persistence on. “SpaceFactory” is set to the fully qualified name of the class that allows us to get the initial connection factory from the message queue. In the example above this property is set to a class that works with Progress Software’s SonicMQ implementation.

During start-up each space installs itself in the CORBA Name Service using its name property as the binding name and in the CORBA Trader Service with the name, persistence, and filter properties. Thus interested applications/processes can find a space by using a well-known name from the CORBA Name Service, or using the space properties from the CORBA Trader Service. For example, an application that wants filtering but is not interested in persistence can indicate these requirements to the CORBA Trader Service, which will then provide the application with a list of CORBA space references that match these requirements. The application may then choose one from that list based on some further screening.

Our implementation of the space gains all of its persistence and filtering capabilities from the underlying messaging queue provider. Our space is the only client of the message queue. Note that in our implementation the only purpose the message queue serves is as a high quality storage/retrieval mechanism that also provides filtering capabilities. We are not relying on the queuing facilities per se.

Let’s describe each method of the CORBA interface in detail now.

`write`
This method is called by an application when it wants to put a stream of bytes into the space and does not want to attach filtering properties to this stream.

`write_filter`
This method is used by an application when it wants to put a stream of bytes into the space and wants to attach filtering properties to this stream. The type FilterSeq evaluates to an array of filters that are attached to that byte stream. A filter is a name-value pair. Hence, a FilterSeq is an array of name value pairs.

`take`
This method is called by an application when it wants to retrieve a stream of bytes from the space. No filtering is performed since none is specified.

`take_filter`
This method is called by an application when it wants to retrieve a stream of bytes from the space. However, in this case a FilterSeq is provided. In order for a match to occur the byte stream must have a subset of the filters provided in the method call and the value of each filter attached to the byte stream must match the value for the corresponding filter in the method call.
`take_filter_as_string`

This method is called by an application when it wants to retrieve a stream of bytes from the space. In this case a string[10] that specifies the exact filter is provided. In order for a match to occur the filter properties attached to the byte stream must satisfy the filter string provided in the method call. This method is used when the filtering conditions cannot be specified as a FilterSeq.

`shutdown`
This method is called to shutdown the space. The shutdown is clean, which means the registration with the Name Service and the Trader Service is removed.

The space implements all methods in the interface as synchronized. Furthermore the take implementations are non-blocking i.e. if there is nothing to take the method returns with nothing.

Conclusion

Distributed applications can be notoriously difficult to design, build, and debug. The distributed environment introduces many complexities, which are not present while writing standalone applications. Some of these challenges include network latency, synchronization and concurrency, and partial failure. Space-based programming, although not a "silver bullet", is an excellent concept that leads towards an elegant solution to these problems. Space-based programming takes us one step further towards achieving our goals in a distributed system, namely those of scalability, high availability, loose coupling, and performance. It also helps us in facing the challenges mentioned above. Best of all, you do not have to buy an expensive implementation to get started with this excellent concept. It's fairly easy to create a homegrown implementation that satisfies your requirements, and it's fun too!

Resources

1. The Linda Group at http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html
2. The JavaSpaces homepage at http://www.javasoft.com/products/javaspaces/
3. IBM, T Spaces at http://www.almaden.ibm.com/cs/TSpaces/.
4. Nicholas J. Carriero. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, Department of Computer Science, 1987.
5. Edward J. Segall. *Tuple Space Operations: Multiple-Key Search, Online Matching and Wait-Free Synchronization*. PhD thesis, Rutgers University, Department of Computer Science, 1993.
6. Gul Agha, et al. *ActorSpaces: An Open Distributed Programming Paradigm*, University of Illinois at Urbana-Champaign, ULIUENG-92-1846.

[10] The syntax of this string is based on a subset of the SQL92 conditional expression syntax. Refer to the "Message Selector Syntax" in the JMS specification for details.