

12.3 JINI

Our next example of a coordination-based system is Jini from Sun Microsystems. Referring to Jini as a coordination-based system is primarily based on its support for generative communication in the form a Linda-like service called **JavaSpaces**, which forms the focus of our discussion of Jini. In addition, it provides support for letting clients easily discover services when they become available, as well as a distributed event and notification system. There are also many other services and provisions that make Jini more than just a coordination-based system. However, the services just mentioned justify its treatment as a case study in coordination-based distributed systems. The specification of Jini is available from Sun Microsystems, and is described in (Sun Microsystems, 2000b; and Waldo, 2000). Keith (2000) gives a good introduction to the core facilities of Jini.

12.3.1 Overview of Jini

Jini is a distributed system that consists of a mixture of different but related elements. It is strongly related to the Java programming language, although many of its principles can be implemented equally well in other languages. An important part of the system is formed by a coordination model for generative communication. We first discuss this model before giving the overall architecture of a typical Jini system.

Coordination Model

Jini provides temporal and referential uncoupling of processes through a Linda-like coordination system called **JavaSpaces** (Freeman et al., 1999; Sun Microsystems, 2000a). A **JavaSpace** is a shared dataspace that stores tuples representing a typed set of references to Java objects. Multiple **JavaSpaces** may coexist in a single Jini system.

Tuples are stored in serialized form. In other words, whenever a process wants to store a tuple, that tuple is first marshaled, implying that all its fields are marshaled as well. As a consequence, when a tuple contains two different fields that refer to the same object, the tuple as stored in a **JavaSpace** implementation will hold two marshaled copies of that object.

A tuple is put into a **JavaSpace** by means of a write operation, which first marshals the tuple before storing it. Each time the write operation is called on a tuple, another marshaled copy of that tuple is stored in the **JavaSpace**, as shown in Fig. 12-2. We will refer to each marshaled copy as a **tuple instance**.

The interesting aspect of generative communication in Jini is the way that tuple instances are read from a **JavaSpace**. To read a tuple instance, a process provides another tuple that it uses as a **template** for matching tuple instances as stored in a **JavaSpace**. Like any other tuple, a template tuple is a typed set of

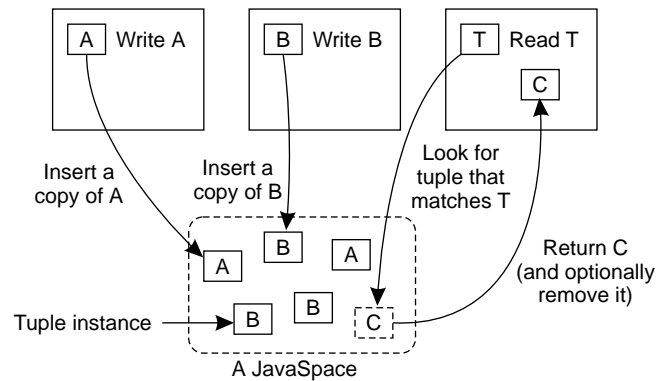


Figure 12-2. The general organization of a JavaSpace in Jini.

object references. Only tuple instances of the same type as the template can be read from a JavaSpace. A field in the template tuple either contains a reference to an actual object or contains the value *NULL*.

To match a tuple instance in a JavaSpace against a template tuple, the latter is marshaled as usual, including its *NULL* fields. For each tuple instance of the same type as the template, a field-by-field comparison is made with the marshaled template tuple. Two fields match if they both have a copy of the same reference or if the field in the template tuple is *NULL*. A tuple instance matches a template tuple if there is a pairwise matching of their respective fields.

When a tuple instance is found that matches the template tuple provided as part of a read operation, that tuple instance is unmarshaled and returned to the reading process. There is also a *take* operation that additionally removes the tuple instance from the JavaSpace. Both operations block the caller until a matching tuple instance is found. It is possible to specify a maximum blocking time. In addition, there are variants that simply return immediately if no matching tuple existed.

Compared to the publish/subscribe model adopted by TIB/Rendezvous, processes that make use of JavaSpaces need not coexist at the same time. In fact, if a JavaSpace is implemented using persistent storage, a complete Jini system can be brought down and later restarted without losing a single tuple instance. Unfortunately, there is also a price to be paid when adopting generative communication. As it turns out, developing highly efficient implementations of a JavaSpace, or any Linda-like system for that matter, is not easy. Generative communication will, in general, impose scalability problems when the model needs to be implemented in a wide-area network. We return to this problem below.

Architecture

JavaSpaces form only part of a Jini system. Like TIB/Rendezvous, Jini is aimed at providing a small, useful set of facilities and services that will allow the construction of distributed applications. A distributed application using Jini is often described as a loose federation of devices, processes, and services. All communication in current Jini systems is based on Java RMI.

The architecture of a Jini system can be viewed in terms of three layers, as shown in Fig. 12-3. The lowest layer is formed by the Jini infrastructure. This layer provides the core facilities of Jini, notably those that enable communication through Java RMI. An important aspect within Jini's model is that clients can easily find services. Services can be provided by regular processes, but may also be provided by hardware devices that cannot run the Jini software, notably the Java virtual machine. Registering and finding services therefore also belong to the Jini infrastructure.

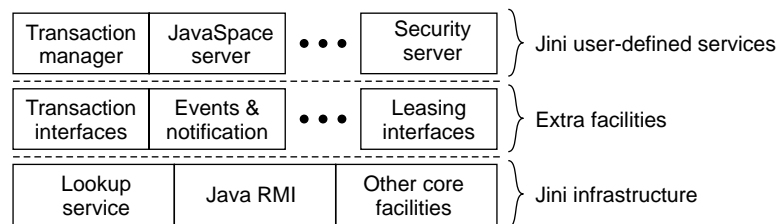


Figure 12-3. The layered architecture of a Jini system.

The second layer is formed by a collection of general-purpose facilities that extend the basic infrastructure, and which can be used to implement various services more effectively. The facilities currently consist of an event and notification subsystem, facilities for associating leases with resources, and the specification of standard interfaces to enable transactions.

The highest layer consists of clients and services. In contrast to the other two layers, Jini does not dictate what should be contained in this layer. At present, it provides a number of services, among which are a JavaSpace server and a transaction manager implementing the Jini transaction interfaces. Programs at the highest layer will generally make direct use also of facilities provided by the Jini infrastructure.

In the following sections we take a closer look at Jini by considering each of the seven principles underlying distributed systems that have been discussed in the first part of this book. In doing so, we concentrate on how these principles relate to Jini as a coordination-based system in which JavaSpaces play a key role.

12.3.2 Communication

As we mentioned, the core communication facilities in Jini are all based on Java RMI. We discussed RMI in Chap. 2, so we will not repeat that here. Besides the generative communication inherent to the JavaSpace model as we discussed above, Jini provides a simple event and notification subsystem as part of its communication facilities, which we consider next.

Events

Jini's event model is relatively simple. If an object has events that may be of interest to clients, a client can register itself with that object. Registering for an event achieves the effect that when the event occurs, the registered client will be notified by the object. Alternatively, the client can tell the object to pass the notification to another process. In either case, the object is passed a remote reference to a **listener object** that can be called back when the event occurs. The callback takes place by means of a Java RMI.

Registration is always subject to a lease. When that lease expires, no more notifications are sent to the registered client (or the process to which the notifications were sent on behalf of the client). Using leases prevents registrations from lasting forever, for example, because the registering client has crashed. Leases are discussed below.

Notification of an event takes place through a remote call by the object to the listener object that was registered for that event. The listener object is invoked again on the next occurrence of the event. Because Jini by itself provides no guarantees that notifications are delivered in the order that events occur, a notification will often carry a sequence number to indicate to the listener object the relative order of the events.

Events can also be used in a JavaSpace. In particular, a client can request to be notified when a specific tuple instance is written to the JavaSpace. In that case, a client calls the notify operation that is implemented by each JavaSpace. This operation takes a template tuple as input that is used to match against stored tuple instances, just like its use in the case of a read or take operation. Using events in combination with JavaSpaces is illustrated in Fig. 12-4. Note that by the time a client is notified of a tuple instance and attempts to read that instance, another process may have already read and removed the tuple instance from the JavaSpace. Such race conditions often happen with generative communication, and are generally hard to avoid.

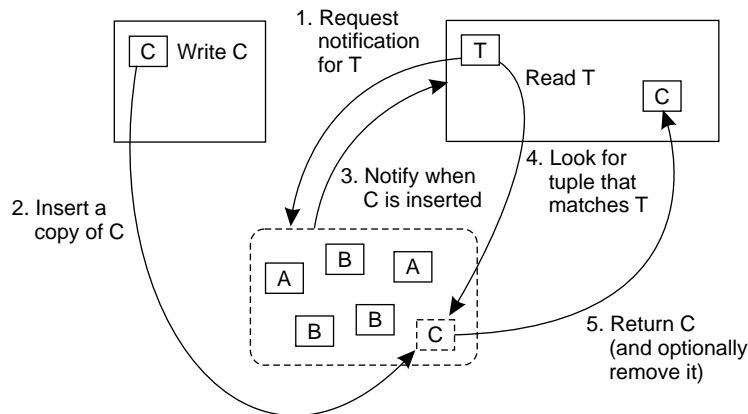


Figure 12-4. Using events in combination with a JavaSpace.

12.3.3 Processes

There is nothing really special about the processes used in a Jini system. However, the implementation of a JavaSpace server often requires special attention. Let us briefly take a look at some of the implementation issues. We concentrate on possible distributed implementations of a JavaSpace server, that is, an implementation by which the collection of tuple instances may be distributed across several machines. A recent overview of implementation techniques for tuple-based runtime systems is given by Rowstron (2001).

An efficient distributed implementation of a JavaSpace has to solve two problems:

1. How to simulate associative addressing without massive searching.
2. How to distribute tuples instances among machines and locate them later.

The key to both problems is to observe that each tuple is a typed data structure. Splitting the tuple space into subspaces, each of whose tuples is of the same type simplifies programming and makes certain optimizations possible. For example, because tuples are typed, it becomes possible to determine at compile time which subspace a call to a write, read, or take operates on. This partitioning means that only a fraction of the set of tuple instances has to be searched.

In addition, each subspace can be organized as a hash table using (part of) its *i*th tuple field as the hash key. Recall that every field in a tuple instance is a marshaled reference to an object. Jini does not prescribe how marshaling should

be done. Therefore, an implementation may decide to marshal a reference in such a way that the first few bytes are used as an identifier of the type of the object that is being marshaled. A call to a write, read, or take operation can then be executed by computing the hash function of the *i*th field to find the position in the table where the tuple instance belongs. Knowing the subspace and table position eliminates all searching. Of course, if the *i*th field of a read or take operation is *NULL*, hashing is not possible, so a complete search of the subspace is generally needed. By carefully choosing the field to hash on, however, searching can often be avoided.

Additional optimizations are also used. For example, the hashing scheme described above distributes the tuples of a given subspace into bins to restrict searching to a single bin. It is possible to place different bins on different machines, both to spread the load more widely and to take advantage of locality. If the hashing function is the type identifier modulo the number of machines, the number of bins scales linearly with the system size (see also Bjornson, 1993).

Now let us briefly examine various implementation techniques for different kinds of hardware. On a multiprocessor, the tuple subspaces can be implemented as hash tables in global memory, one for each subspace. When a JavaSpace operation is performed, the corresponding subspace is locked, the tuple instance is entered or removed, and the subspace unlocked.

On a multicomputer, the best choice depends on the communication architecture. If reliable broadcasting is available, a serious candidate is to replicate all the subspaces in full on all machines, as shown in Fig. 12-5. When a write is done, the new tuple instance is broadcast and entered into the appropriate subspace on each machine. To do a read or take operation, the local subspace is searched. However, since successful completion of a take requires removing the tuple instance from the JavaSpace, a delete protocol is required to remove it from all machines. To prevent race conditions and deadlocks, a two-phase commit protocol can be used.

This design is straightforward, but may not scale well as the system grows in the number of tuple instances and the size of the network. For example, implementing this scheme across a wide-area network is prohibitively expensive.

The inverse design is to do *WRITES* locally, storing the tuple instance only on the machine that generated it, as shown in Fig. 12-6. To do a read or take, a process must broadcast the template tuple. Each recipient then checks to see if it has a match, sending back a reply if it does.

If the tuple instance is not present, or if the broadcast is not received at the machine holding the tuple, the requesting machine retransmits the broadcast request ad infinitum, increasing the interval between broadcasts until a suitable tuple instance materializes and the request can be satisfied. If two or more tuple instances are sent, they are treated like local writes and the instances are effectively moved from the machines that had them to the one doing the request. In fact, the runtime system can even move tuples around on its own to balance the

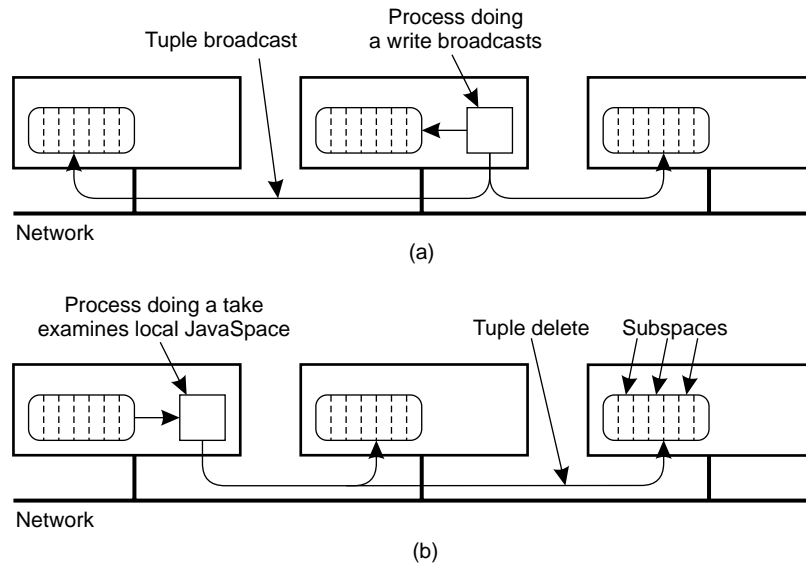


Figure 12-5. A JavaSpace can be replicated on all machines. The dotted lines show the partitioning of the JavaSpace into subspaces. (a) Tuples are broadcast on *WRITE*. (b) *READS* are local, but the removing an instance when calling *TAKE* must be broadcast.

load. Carriero and Gelernter (1986) used this method for implementing the Linda tuple space on a LAN.

These two methods can be combined to produce a system with partial replication. As a simple example, imagine that all the machines logically form a rectangular grid, as shown in Fig. 12-7. When a process on a machine *A* wants to do a write, it broadcasts (or sends by point-to-point message) the tuple to all machines in its row of the grid. When a process on a machine *B* wants to read or take a tuple instance, it broadcasts the template tuple to all machines in its column. Due to the geometry, there will always be exactly one machine that sees both the tuple instance and the template tuple (*C* in this example), and that machine makes the match and sends the tuple instance to the process requesting for it. This approach is similar to using quorum-based replication as we discussed in Chap. 6. It has been used to implement Linda (Ahuja et al., 1988), but also to implement tuple spaces on a cluster (Tolksdorf, 1995)

The implementations discussed so far have serious scalability problems caused by the fact that multicasting is needed either to insert a tuple into a tuple space, or to remove one. Wide-area implementations of tuple spaces do not exist. At best, several *different* tuple spaces can coexist in a single system, where each tuple space itself is implemented on a single server or on a local-area network. This approach is used, for example, in PageSpaces (Ciancarini et al., 1998) and

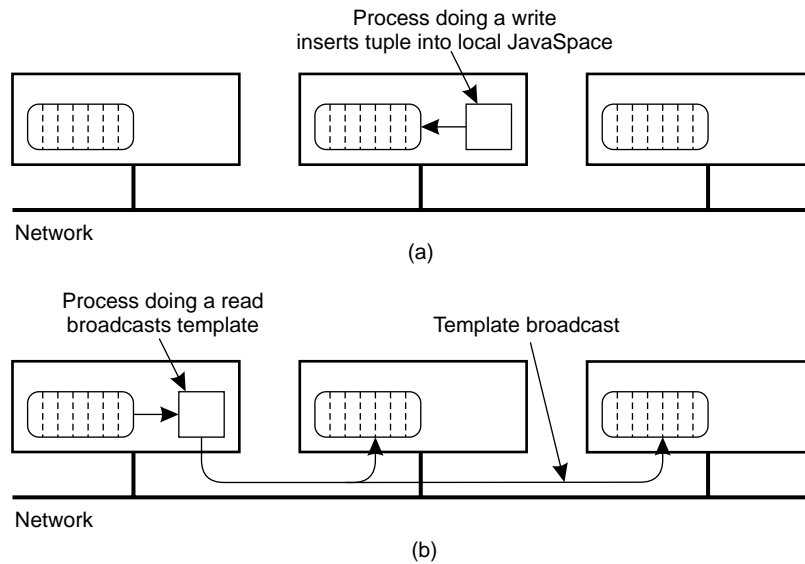


Figure 12-6. Unreplicated JavaSpace. (a) A *WRITE* is done locally. (b) A *READ* or *TAKE* requires the template tuple to be broadcast in order to find a tuple instance.

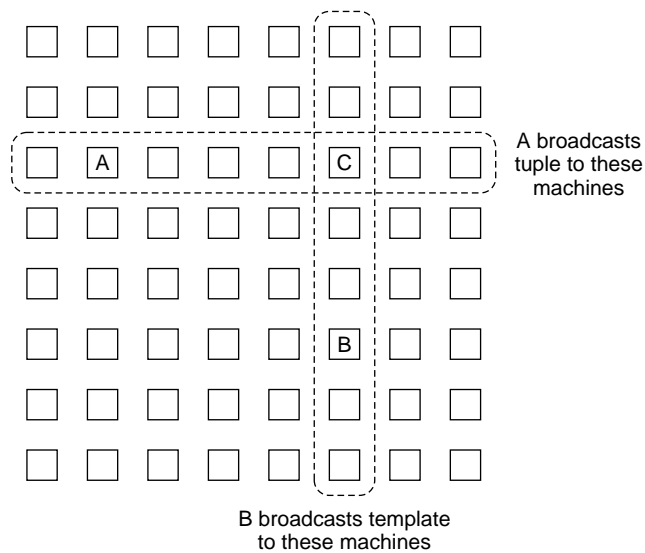


Figure 12-7. Partial broadcasting of tuples and template tuples.

WCL (Rowstron and Wray, 1998). In WCL, each tuple-space server is responsible for an entire tuple space. In other words, a process will always be directed to exactly one server. However, it is possible to migrate a tuple space to a different server to enhance performance. How to develop an efficient wide-area implementation of tuple spaces is still an open question.

12.3.4 Naming

Jini does not provide a traditional naming service like those found in object-based distributed systems or distributed file systems. Such a naming service can easily be implemented as part of the service layer within the Jini architecture, but as such does not form part of Jini's core. However, Jini does provide a service that allows clients to look up registered services using an attribute-based search facility. This lookup service is described next.

The Jini Lookup Service

As we mentioned, one of the design goals of Jini is to provide a system that easily allows clients to look up new services as they come available. In principle, a JavaSpace can implement such a service. Whenever a service is added to an existing system, it inserts a tuple instance into a JavaSpace describing itself. Clients that are looking for specific services request the JavaSpace to be notified when a service inserts a tuple that matches what the client is looking for.

Instead of using a JavaSpace, Jini provides a separate, specialized **lookup service** as part of its lowest-level infrastructure layer, as shown in Fig. 12-3. A service registers itself by providing a set of (*attribute, value*)-pairs that describe, for instance, what the service has to offer, and where it can be contacted. A client can look for a service by providing a template to the lookup service, similar to the template tuples in a JavaSpace. The lookup service returns information on matching services. Let us briefly examine what a lookup service looks like.

Each service has an associated **service identifier**, which is a globally unique 128-bit value generated by the lookup service. A service uses this identifier to register a **service item** at the lookup service. A service item can be viewed as a record with three fields as shown in Fig. 12-8.

Field	Description
ServiceID	The identifier of the service associated with this item.
Service	A (possibly remote) reference to the object implementing the service.
AttributeSets	A set of tuples describing the service.

Figure 12-8. The organization of a service item.

The *ServiceID* field contains the service identifier assigned to the service by

the lookup service. The identifier operates as a unique key in the lookup service. There will thus never be two service items stored in the lookup service having the same identifier.

The *Service* field contains a reference to an object. In many cases, it will be a reference to a remote object, implying that the client obtaining this reference from the lookup service can immediately invoke that object using Java's RMI. Recall from Chap. 2 that a remote reference in Java is often implemented as a marshaled proxy that merely needs to be unmarshaled to allow its holder to invoke the object.

The *AttributeSets* field is a set of tuples, similar to those used in a JavaSpace. Each tuple essentially corresponds to a Java object with each field in the tuple describing an (*attribute, value*)-pair of that object. Using the same technique as in JavaSpaces, a client can provide a template tuple when looking for specific tuple instances. The lookup service will select only those tuples that match the template.

As in the case of JavaSpaces, a client can request the lookup service to send a notification when a service item is inserted that matches the client's template tuple.

To assist in setting up a Jini system, there are a number of predefined tuples that can be used to register services, as shown in Fig. 12-9. For these tuples, all attributes are represented as character strings, but other representations are allowed, thus providing as much flexibility as needed.

Tuple type	Attributes
ServiceInfo	Name, manufacturer, vendor, version, model, serial number
Location	Floor, room, building
Address	Street, organization, organizational unit, locality, state or province, postal code, country

Figure 12-9. Examples of predefined tuples for service items.

We have implicitly assumed that there is only a single lookup service. However, Jini allows several lookup services to coexist. Each service may be responsible for a group of services. In this way, the load on a single lookup service can be distributed across different machines.

In our description thus far, there is one important issue missing, namely how is a lookup service looked up? A standard approach in many distributed systems is to configure a lookup server with a well-known address. Jini takes a different approach, and lets a client multicast a message requesting lookup services to tell where they are located. Without taking special measures, this approach works efficiently only for local-area networks.

In addition, lookup services will regularly announce their presence also using multicasting. A client can then register the location of a lookup service for the

next time it wants to search for a specific service. The exact details of the associated protocols are described in (Waldo, 2000), which also contains exact specifications on interfaces to lookup services.

Leasing

Related to naming issues is the management of references to objects. As we explained in Chap. 4 an approach to managing references is to let a referenced object keep track of who is referring to it, leading to what are known as reference lists. To keep the list short but also to handle situations in which a referring processes crashes, it is convenient to make use of leases. When a lease expires, a reference becomes invalid and is removed from the object's reference list. When that list becomes empty, the object can safely destroy itself.

Jini makes extensive use of leases to ensure that objects are cleaned up when they are no longer referred to. For example, whenever a process writes a tuple to a JavaSpace, it is returned a lease specifying how long that tuple will be stored until it is destroyed. In this case, the write operation allows the caller to specify a required lease period.

Leases are never handed out with ironclad guarantees. Instead, when a process hands out a lease, it effectively promises to do its best to keep the object associated with the lease for at least the time specified. A process holding a lease can always request the lease to be renewed. However, it is up to the leaser to decide whether a renewal is granted or not.

Interfaces for leases have been standardized. Whenever a lease is used in Jini, it follows the same specification everywhere. Details on these interfaces can be found in (Waldo, 2000).

12.3.5 Synchronization

Jini provides a few synchronization mechanisms. One important class of mechanisms is implemented as part of a JavaSpace, namely the blocking operations read and take. These operations can be used to express many different synchronization patterns as illustrated for parallel programs in (Carriero and Gelernter, 1989). In addition to synchronization through these two operations, Jini provides a notion of transactions as well, which we discuss next.

Transactions

To assist in carrying out a series of operations on multiple objects, Jini supports transactions that use a two-phase commit protocol. This support is essentially given only in the form of a set of interfaces; their actual implementation is left to others. However, Jini can be configured with a default transaction manager.

This approach has an important implication, namely that the ACID properties

of a transaction are not provided by Jini itself. Instead, it is assumed that these properties are jointly implemented by the various processes that take part in a transaction. However, the interaction between the processes adheres to the pattern in transactions using two-phase commit.

The overall model of a transaction in Jini is shown in Fig. 12-10. A client can start a transaction by issuing a request to a transaction manager, which will return a transaction identifier. As in many other occasions, the client is required to specify how long it will take before the transaction aborts or commits. Additionally, the manager will hand out a lease for the newly created transaction, and in all cases abort the transaction when the lease expires.

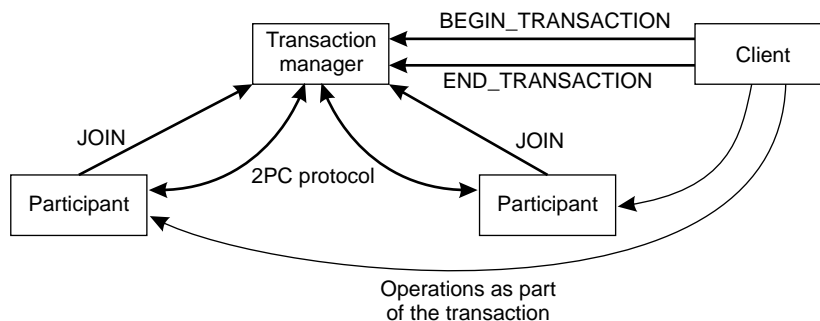


Figure 12-10. The general organization of a transaction in Jini. Thick lines show communication as required by Jini's transaction protocol.

The client can instruct other processes to join the transaction. These processes are required to implement a predefined interface that will allow the transaction manager to further control the transaction. This interface contains operations such as commit and abort that are invoked by the transaction manager to tell a participant what to do. It is up to the participant to correctly implement these methods.

Assuming that a transaction can be finished before its associated lease expires, the client will tell the transaction manager that it should commit or abort the transaction. From there on, the transaction manager executes the two-phase commit protocol and reports the result to the client. The actual protocol has been covered in detail in Chap. 7 and will not be repeated here.

Jini also supports nested transactions. In that case, a transaction manager is requested to start a transaction as part of an existing transaction. Again, the client is in control of organizing the transaction, that is, it determines which processes should join a transaction. In the case of nested transactions, processes are requested to join specific subtransactions.

A JavaSpace can participate in a transaction as well. A transaction, in turn, may span multiple JavaSpaces. Whenever a process issues a JavaSpace operation, it can pass along a transaction identifier. If the JavaSpace had not yet joined the indicated transaction, it will do that first. The JavaSpace servers along with the

transaction manager jointly ensure that the ACID properties are preserved. In addition, the Jini implementation of JavaSpaces, in combination with the default transaction manager provided as part of Jini, apply strict two-phase locking to avoid cascaded aborts. We explained two-phase locking in Chap. 5.

12.3.6 Caching and Replication

As in TIB/Rendezvous no special measures are provided by Jini for caching or replication. These matters are entirely left to the applications that are built as part of a Jini-based system. The only place where Jini assumes that services may be replicated for fault tolerance is with respect to its lookup services.

12.3.7 Fault Tolerance

Jini by itself does not provide additional support for fault tolerance except for a transaction manager that implements the transaction protocol described above. Jini expects that components that use Jini as a basis implement their own fault-tolerance measures as needed.

Much research has been done on incorporating fault tolerance in the original Linda tuple spaces (which form the basis for JavaSpaces). For example, Bakken and Schlichting (1995) describe an approach based on active replication of tuple spaces. In addition, they extend the programming model to group multiple operations into a single atomic unit. An approach more in line with that of Jini in which tuple space operations are grouped into transactions is described in (Shasha and Jeong, 1994).

With respect to communication, note that virtually all communication is done by means of Java RMI, which itself is generally implemented using a reliable, lower-level communication protocol such as HTTP or TCP.

12.3.8 Security

Security in Jini relies entirely on the security provided by Java RMI. Many of the issues in Java RMI have already been discussed in Chap. 8, notably with respect to protecting against dynamically downloaded code by means of stack introspection. Recall that an important property of stack introspection was the possibility to attach access privileges to classes. These privileges can be checked at runtime using the Java security manager.

What has been added to Jini is a separate service called the **Java Authentication and Authorization Service (JAAS)** that handles *user* authentication and authorization in Java-based systems such as Jini. Following an approach similar to other distributed systems, JAAS separates the interface it offers to users for authentication and access control from the actual implementation of those services. This separation is achieved through **PAM**, the **Pluggable Authentication**

Module (Samar and Lai, 1996). PAM essentially provides an intermediate layer between applications and security services, providing a standard interface to both groups, as shown in Fig. 12-11. JAAS is a Java implementation of PAM.

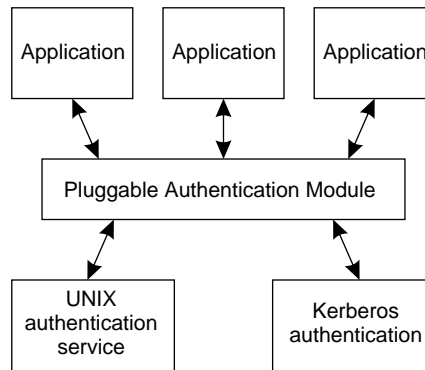


Figure 12-11. The position of PAM with respect to security services.

What JAAS adds to Java's existing access control mechanisms is the facility to perform access control with respect to previously authenticated users. Recall that Java as explained in Chap. 8, provides facilities to do class-based access control by which privileges are associated to classes. JAAS can also handle users. In effect, JAAS provides necessary support for enabling a Jini-based distributed system to support multiple users.

As JAAS is similar to the authentication and authorization services we have discussed so far, we will not go into further details. More information can be found in (Lai et al., 1999).