Final Project Report: JavaSpaces Analysis

Stefan Witt
WSU ID 10539995
stefan_witt@wsu.edu

CptS 562
May 9, 2002

1 Introduction

For programming distributed systems it is very convenient to have shared memory that can be accessed by cooperating processes. The abstraction of a commonly usable storage greatly simplifies programming a distributed system, because the developer does not need to bother about the difficulties that are inherent in the distribution, such as message ordering or syncrony of memory accesses. Distributed shared memory provides temporal and spatial decoupling of the cooperating processes: The processes do not need to have overlapping lifetimes, and they do not need to know each other's identities. These principles will be discussed later in this project report.

One instance of distributed shared memory are tuple spaces. A tuple space is an unordered bag of tuples, which can be accessed associatively. A tuple is – like a data structure – a composite data type that consists of elementary data types, such as integers or strings, so it is basically the same as a mathematical tuple. In a tuple space arbitrary tuples can be stored, i.e. the space is not limited to tuples of the same type. Like stated above, a tuple space is content addressable memory. Tuples can be stored in the space, and they can be taken from the tuple space by using pattern matching. For this pattern matching any tuple that matches a specified template is taken from the tuple space. It is important to note that storing and removing tuples are the only operations that are allowed on tuple spaces, and the modification of a tuple within the space is not possible.

Examples of tuple spaces are Linda, FT-Linda, PLinda, and JavaSpaces. The concept of tuple spaces was first introduced with the Linda programming and coordination language [Gel85]. It was designed for distributed parallel programs, i.e. replicated workers that work on partial problems of a task and are coordinated by a master. FT-Linda and PLinda are extensions of Linda that augment Linda primitives with fault tolerant extensions. Linda's problem is that it cannot handle failed processes, such as failed workers or a failed master. Tuples can be lost if they are withdrawn from the space by a process and the process crahes right after that. FT-Linda and PLinda provide means to remedy this problem. JavaSpaces [FHA99] is one of the services provided by Jini [OW00]. It allows to hold Java objects with methods and attributes as entries in the tuple space, which is called JavaSpace.

This report is organized as follows. Section 2 introduces the different tuple spaces in more detail. Section 3 describes fault tolerance aspects of JavaSpaces. In the main part that consists of sections 4 and 5 presents an example application for JavaSpaces and analyses its performance. Finally, 6 wraps up the results.

2 Tuple Spaces

This section presents different tuple spaces. The JavaSpaces tuple space is most important in this context, because it is analysed in the following sections.

2.1 Linda

The coordination language Linda was first introduced in [Gel85]. The tuples in Linda consist of a type name and a list of constants, which are the actual tuple values. Three primitives are originally provided by Linda: `out`, `in`, and `read`.

The primitive `out(N, P`$_2$`, ..., P`$_j$`)` deposits the specified tuple with name N and values $P_2, \ldots, P_j$ into the tuple space. The name is a logical name, i.e. it does not need to be unique, thus multiple tuples may have the same name.

Symmetrically to `out`, the primitive `in(N, P`$_2$`, ..., P`$_j$`)` withdraws a tuple of the name N from the tuple space. The parameters $P_2, \ldots, P_j$ are bound with the corresponding values of the withdrawn tuple. Optionally, a pattern matching can be performed; this is done by specifying values instead of variables as parameters of `in`. In Linda implementations the `in` operation is blocking, i.e. if there is no matching tuple of the specified type, the caller blocks and waits for one. If there is more than one process waiting for a tuple of a specific type, only one of them gets it, i.e. no tuples are duplicated. Furthermore, there is a non-blocking version of `in` called `inp`. It only withdraws a tuple of a specific type if it exists in the tuple space, otherwise it returns non-success.

The `read` statement with the syntax `read(N, P`$_2$`, ..., P`$_j$`)` is the same as `in`, except that the tuple remains in the tuple space instead of being withdrawn. Again, there is a non-blocking version of `read` called `readp`.

With these three primitives Linda is a simple, but powerful coordination language for accessing the shared tuple space.

2.2 FT-Linda

The coordination language Linda, which has been introduced in the previous subsection, has a fundamental drawback: In the case of process failures tuples can accidentally be lost or duplicated. For the first case, consider a replicated worker that has withdrawn a task description tuple from the tuple space, but it crashes before it completes the task by writing the result back to the tuple space. Then the tuple that describes the task is lost and cannot be recovered. Even a recovery mechanism in the replicated worker does not help, because there is always a small window of vulnerability between the withdrawal and writing the tuple to a local persistent storage. In the case of a crashing master a task description tuple can be duplicated, if the master does not recover information about the task description that it has deposited in the tuple space. Again, this can happen, because there is a window of vulnerability between the deposition of the task description in the tuple space and the writing of the tuple to local

persistent storage; note that the master cannot rely on finding its deposited tuples in the tuple space to determine whether it has already created a task, because a worker might have withdrawn the tuple in order to process it.

To remedy there drawbacks of Linda, a fault tolerant version of Linda has been developed, FT-Linda [BS95]. It uses two basic improvements, namely stable tuple spaces and atomic execution of tuple space operations. The former enhancement is done using fault tolerance techniques, i.e. replication of the tuple space on n replicas in order to survive $n - 1$ crash failures. The fault model that is assumed is fail-silent, i.e. the replicas are either correct in the time and value dimension, or they crash benignly without sending corrupted or incorrect values. A state machine approach is used for consistency and coordination among the replicas. For details on the realization see [BS95].

The second enhancement, the atomic execution of tuple space operations, is done using *atomic guarded statements* (AGS). An AGS is a construct of the form `<guard → body>` with the following semantics: The atomic guarded statement blocks until the guard evaluates to true or false, where `guard` can be any tuple space operation like `in` or `out`, or it can be a boolean value. If the guard is true, then the body is executed atomically, i.e. on an all-or-none basis. The body may be a series of instructions `inp`, `out`, `readp`, and other specialized instructions such as `copy`. In [BS95] there is shown how to design a fault-tolerant replicated worker pattern using FT-Linda. This FT-Linda version of the replicated worker does not have the drawbacks of the corresponding Linda version – the windows of vulnerability are eliminated.

2.3 PLinda

PLinda is a persistent version of Linda, whence the name. As described in [BJL+97], PLinda augments Linda by resilient processes and fault tolerant tuple spaces.

In order to make processes resilient to failures, the concept of transactions is introduced. That means a sequence of operations on the tuple space is performed atomically on an all or none basis. The `xstart` operation initiates a transaction, and the `xcommit` operation commits it. PLinda ensures atomic execution of all tuple space operations between `xstart` and `xcommit`. With transactions, the window of vulnerability of tuple space accesses is removed, and therefore, the tuple space is always in a correct state, if fail-silent processes are assumed, i.e. processes that do not deposit erroneous tuples in the tuple space.

The second enhancement in PLinda are fault tolerant tuple spaces. PLinda uses periodic checkpointing to achieve this. At a checkpoint PLinda saves the entire tuple space to persistent storage, and it ensures to save only committed transactions so that the saved tuple space is in a consistent state. It is interesting to note that PLinda does not use any logging replay mechanism to recover from a tuple space failure. Instead, when a tuple failure recovery is done, the PLinda system kills all active processes and respawns them using its process resilience mechanisms. The restarted processes will continue from the state at which they committed the last transaction that was checkpointed to persistent storage, i.e. the replay is performed by the processes that use the tuple space.

Another fault tolerance mechanism that is employed in PLinda is failure detection of processes that use the tuple space. PLinda assumes a synchronous cooperation model and uses a timeout mechanism for failure detection: If a processes does not access the tuple space for a specific period of time, it is assumed to be failed. Then the system stops the suspected process and respawns a new backup process in place of the failed one. If the failure detection was wrong, i.e. it suspected a process to be failed,

but actually a link failure such as a network partition occured, then it will stop the suspected process as soon as it is reachable again. This is due to inconsistencies that can occur because the backup process is performing the same task as the accidentally suspected process.

Since PLinda can be used for parallel programming with replicated workers, it can be used in a network of workstations, as described in [BJL+97] and [STW97]. The main idea is to use the idle time of workstations in a local area network for running replicated workers that compute tasks they obtain from a tuple space. The workstation program detects idle time and runs only if the CPU computing time is not used by any other user process.

2.4 JavaSpaces

Another instance of tuple spaces is JavaSpaces [FHA99], which is one of the Jini services. Jini is presented in [OW00]; it is middleware for the Java programming language, and it allows the programming of a dynamic distributed system, the Jini community. A user process that wishes to use Jini services first finds a lookup service, which is then used to find services within the community. The user process obtains a service interface as a local object, which is a local stub for the service. The methods of the local object are then used to invoke service primitives. Thus, the principle is very much like CORBA (Common Object Request Broker Architecture), which also uses local stubs for remote access of objects.

Like stated above, one of the services Jini provides is JavaSpaces. Unlike the tuple spaces introduced in the previous subsections, JavaSpaces allows the storage of objects (called *entries*) in the space instead of tuples. In this context objects means Java objects that maintain a state and provide methods for operations on them. Thus, code shipping to replicated workers is possible using JavaSpaces. This project focuses on JavaSpaces, which is further presented in the next section. An example application of replicated workers is given in section 4.

3 Fault Tolerant Aspects of JavaSpaces

In order to tolerate faults, JavaSpaces uses different means. Fault tolerance is very important for JavaSpaces, because JavaSpace applications are always distributed.

3.1 Jini Fault Tolerance

As described in section 2.4, the JavaSpaces service is part of Jini. Jini has been developed to be fault tolerant in the sense that crashing servers can recover automatically. This is done using the so-called RMI activation (Remote Method Invocation). Each Jini service that is started registers itself at the RMI daemon `rmid`. `rmid` is a program that keeps track of activatable services, and it is used for RMI, on which Jini is based upon. An important feature is that activatable services are persistent, i.e after a system crash `rmid` tries to restart them automatically so that no explicit restart of each service is required – restarting `rmid` is sufficient. To accomplish this, `rmid` keeps track of the service using a log-file. However, it does not log the internal state of the services, and therefore, still data may be lost when the system crashes.

Each service has to maintain a log-file in order to recover from a system failure. The Jini services that JavaSpaces needs to run all use logging: the lookup service `reggie`, and the transaction manager

`mahalo`. The JavaSpaces service called `outrigger` itself provides two alternatives: A persistent version and a non-persistent version. While the non-persistent version does not survive a system failure, the persistent version can recover from its log-file after a crash. For more details on `rmid` and the Jini services refer to [OW00].

3.2 JavaSpaces Transactions

In order to avoid lost or duplicated tuples as in the scenarois described in section 2.2, JavaSpaces provides a transaction mechanism. It guarantees the ACID properties atomicity, consistency, isolation, and durability. Transactions consist of a number of space accesses with the operations `take`, `read`, `write`, etc. However, the semantics of space operations in a transaction is slightly different from the semantics outside a transaction.

The main differences are (for more details see [FHA99]):

- Entries that are written within a transaction are only visible within the transaction. If the object is taken out of the space again within the transaction, it is never seen outside the transaction.
- A `read` operation can match either an entry written under the transaction or an entry in the space. Which one `read` returns, is not defined.
- If an entry is read from the space or taken out of the space during a transaction, other transactions are kept from reading that particular entry. This ensures the consistency of the space, because other transactions cannot change the entry by reading and modifying it. If another transaction issues a `read`, then the `read` blocks until a matching entry is found, i.e. it would wait until the first transaction commits. However, the semantics for `readIfExists` is slightly different: If there is no matching entry is found in the space, then it can wait until a transaction commits with a matching entry.

4 Replicated Worker Example

In this chapter an example JavaSpaces application is presented that follows the replicated worker pattern. It is a program that does a known-plaintext attack on data that is encrypted using a 4-bit RC4 cipher. This is a nice example for a program that can be executed at an arbitrary degree of parallelism.

4.1 The RC4 Stream Cipher

RC4 is a random number generator that has been developed by Ron Rivest[1]. It is described in [Schn96], and a detailed analysis can be found in [HW01]. The generation of random numbers with RC4 is very efficient, and therefore, this generator can be used as a stream cipher. Usually, it uses a width of 8 or 16 bits, but here, we reduce it to 4 bits because we emphasize on the replicated worker rather than on the security aspect. In the following text the RC4 generator is briefly introduced, following [HW01].

For the initialization of RC4 a key is needed. It consists of 2^4 values $K_0, \ldots, K_{15}$, which can have values within $0, \ldots, 15$. The algorithm uses an S-box (substitution box) of 2^4 values $S_0, \ldots, S_{15}$, which

[1] RC4 is a patent of RSA Data Security Inc., and the usage outside University needs a license.

are out of the range $0, \ldots, 15$. It is initialized with $S_0 = 0, S_1 = 1, \ldots, S_{15} = 15$. RC4 uses two registers i and j. Then the following instructions are executed:

$j \leftarrow 0$
for $i = 0$ **to** 15 **do**
  $j \leftarrow (j + S_i + K_i) \bmod 15$
  swap S_i und S_j
end for
$i \leftarrow 0$
$j \leftarrow 0$

This permutes the S-box 16 times and yields the initial state of the random number generator. In order to generate a random number, the following instructions are executed:

$i \leftarrow (i + 1) \bmod 15$
$j \leftarrow (j + S_i) \bmod 15$
swap S_i und S_j
$t \leftarrow (S_i + S_j) \bmod 15$
output S_t

That is the RC4 algorithm, which is indeed very simple, but very powerful. It is extremely fast and secure, i.e. there is no known attack to break RC4 [Schn96]. The 4-bit version of RC4 that has been introduced above has exactly $5\ 356\ 234\ 211\ 328\ 000$ internal states, i.e. the maximum period consists of this many random numbers [HW01].

The random number generator can easily be used to encrypt data. A key of 16 values of 4-bit width is used to initialize the generator. Then the generator is called twice to create 8 random bits, which are then exclusive-or-combined with the first data byte. The same is then done with the second data byte, and so on.

For this case study this algorithm has been implemented in the programming language Java. The program is `RC4.java`, and it takes an input file and an output file as parameters. Upon execution it prompts for a pass phrase, which is used to generate the key $K_0, \ldots, K_{15}$. This is done as follows: First, the key is initialized as $K_0 = 0, K_1 = 1, \ldots, K_{15} = 15$. Then the lower 4 bits of the ASCII representation of 2 consecutive letters are taken as p and q , and the corresponding values K_p and K_q are swapped. This done for every 2 letters in the pass phrase.

Note that the program can also be used to decrypt data, since the RC4 cipher is symmetric.

4.2 Sequential Implementation

One approach to break the RC4 cipher is to do a known-plaintext attack on the encrypted data. So the attack needs the encrypted data as input as well as a known plaintext that appears in the decrypted data, for instance an encrypted letter and “hello”.

The break algorithm uses a brute force attack. It generates all permutations of the numbers $0, 1, \ldots, 15$, and it uses each permutation as a key for RC4, decrypts the data and searches for the plain text. This algorithm uses exponential time, since the number of permutations is $15!$, which is of exponential order.

Therefore, it is very inefficient, but still it is the only officially known attacks against the RC4 cipher.

This algorithm has been implemented in a sequential program called `RC4Breaker.java`. It takes the encrypted file and an output file for the decrypted data as parameters. Then it prompts for the known plaintext.

4.3 Implementation With JavaSpaces

Since the sequential implementation is very inefficient it is imperative to modify it to a parallel version. This is possible in principle, because the algorithm allows for an arbitrary degree of parallelism. Each parallel processor simply tries one key independently from the other processors.

To coordinate the cooperating processes JavaSpaces is used. We have one master, which puts test RC4 keys into the tuple space, and a number of replicated workers, which take keys from the tuple space, try to break the RC4 cipher and write the result (true or false) back to the tuple space. The master collects the results until it finds true, i.e. a matching key.

In more detail, we have two different objects that the master puts into the JavaSpace. First, a `CipherText` object, which simply contains the encrypted data that has to be decrypted. The master puts only one `CipherText` object in the JavaSpace in its initialization phase. It is used to pass the encrypted data to the workers without transmitting it again and again with every key. This is very useful for a low network bandwidth consumption, because the encrypted text might be long. The second object type that the master deposits in the JavaSpace are `RC4Task` objects. They contain 4 fields: `key`, `plainText`, `done`, and `plainTextFound`. The `key` field contains the key that a worker has to try on the cipher text. The `plainText` field holds the known plaintext that has to be found in the decrypted data. The master sets `done` and `plainTextFound` to false to indicate that this key has not been tried yet.

A worker reads the `CipherText` object from the JavaSpace at startup. It only operates on this encrypted data in the future. Then it enters an infinite loop, in which it withdraws `RC4Task` objects from the JavaSpace. It decrypts the data and searches for the plaintext. According to the result, it sets `plainTextFound` in the `RC4Task` object. Finally, the worker sets `done` to true and deposits the `RC4Task` object in the JavaSpace again. All this is done in a single transaction, i.e. the task is executed atomically.

The master collects the `RC4Task` objects from the JavaSpace that have `done` set to true, i.e. finished tasks. If it finds a completed `RC4Task` with `plainTextFound` set to true, it uses the key to decrypt the file and terminates. In order not to overfill the JavaSpace with tasks it synchronizes with the workers by keeping track of the number of tasks that it has put into the JavaSpace. It respects an upper bound of the number of outstanding tasks. This bound can be configured in the master source code.

The key fact in the JavaSpaces implementation of the RC4 cipher breaker is that it can be accelerated arbitrarily, because an arbitrary number of workers can be started to perform tasks that the master writes.

5 JavaSpaces Performance Analysis

In this section the performance of JavaSpaces is analysed. To do this, the example from the previous section 4 is run. It is a canonical example of JavaSpaces, because it addresses the main domain of JavaSpaces, namely coordinating parallel computing.

5.1 Running The Example

In order to run the replicated worker example, the Jini services have to be started first. These are an HTTP-Server in order to allow the Jini services to download code from a server via Remote Method Invocation (for details on this see [OW00]). Next, the RMI activation daemon `rmid` has to be started such that initially there is no log-file, so it does not try to recover from a system crash that never occured. Then the Jini lookup service `reggie`, the transaction manager `mahalo`, and the JavaSpaces service `outrigger` (here the persistent version is chosen) can be started. To accomplish all this, a Linux shell script `javaspaces.sh` has been developed in this project, which does the necessary calls.

To generate an encrypted file, a PDF file[2] of the size 139771 is encrypted with the `RC4.java` program introduced in section 4. The key is `hijklm`, thus it is a rather weak key, which can be broken with 5167 tries (opposed to a maximum of $15!$).

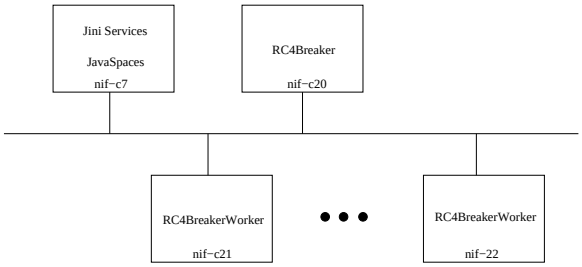

Figure 1: Used network topology.

On a different machine, the `RC4BreakerJS` program is run; the known plain-text is "%PDF", the head of a PDF file. Then the workers are run again on other machines, as shown in figure 1. It is important to run each of these programs on a different machine, because only in this case the maximum performance can be achieved.

5.2 Performance Results

The results for the performance measurements are shown in table 1. The time to break the RC4 cipher is shown as a function of the number of workers `RC4BreakerWorker` and the maximum number of tasks the `RC4BreakerJS` writes to the space.

Number of Workers	Max. Outstanding	Time
1	3	12'11"
2	4	6'47"
2	6	7'15"
3	5	6'19"
3	9	5'54"

Table 1: Time to break RC4 for a number of workers and a maximum number of outstanding requests.

It can be observed that the time to break RC4 depends on the maximum number of outstanding tasks. However, it is not possible to determine the exact relation between them. In table 1, the time increases for an increased number of outstanding taksk for 2 workers. However, for 3 workers, the time decreases for an increased number of outstanding tasks.

5.3 Result Evaluation

The main result of the performance analysis is that the time is not half the time if 2 workers are used instead of one, and it is not one third if 3 workers are used instead of one. Therefore, the time does not scale reciprocal with the number of workers, as it would have been excepted.

There are two reasons for this behavior. First, there is a huge overhead when JavaSpaces is used for coordination. This is an inherent problem of the JavaSpaces implementation; it is not possible to scale the number of outstanding tasks right in order to reach the maximum performance.

But the second reason is more severe. This is a drawback in the infrastructure that has been used when running the example. It lies in the operating system Linux on which JavaSpaces and the JavaSpaces

[2] my literature survey of CS562 ;-)

programs were run. The problem is that Jini heavily uses multi-threading, but since JDK 1.3 each thread is started as a native process. When the JavaSpace service is started, about 80 processes are generated, and the system slows down dramatically[3]. A test showed that this does not happen under Solaris and SunOS. The result for this test, however, is that the JavaSpaces service is unable to handle more than two or even just one worker properly when it runs under Linux. Therefore, the scalability feature of JavaSpaces for parallel computing is not guaranteed. This is the reason why no measurements of more time-consuming RC4 cipher breaking computations have been done in this project.

6 Conclusions

In this project the JavaSpaces tuple space has been presented and analysed. This technology can be used for coordinating parallel computations of similar tasks. As an example, an RC4 cipher breaking program has been implemented. The result is that the performance of JavaSpaces and Jini heavily depends on the operating system that is used. While it it not very performant under Linux, it runs better under Solaris.

References

[BS95] David E. Bakken, Richard D. Schlichting: *Supporting Fault-Tolerant Parallel Programming in Linda*, IEEE Transactions on Parallel and Distributed Systems, March 1995

[BJL+97] Thomas Brown, Karpjoo Jeong, Bin Li, Suren Talla, Peter Wyckoff, Dennis Shasha: *PLinda User Manual*, TR1996-729, December 1996
Available at **`http://csdocs.cs.nyu.edu/Dienst/UI/2.0/Describe/ncstrl.nyu_cs%2FTR1996-729`**.

[FHA99] Eric Freeman, Susanne Hupfer, Ken Arnold: *JavaSpaces Principles, Patterns, and Practice*, Addison Wesley, 1999

[Gel85] David Gelernter: *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, Vol. 7 No. 1 pp. 80–112, January 1985

[HW01] Peter Hartmann, Stefan Witt: *Vergleichende Analyse und VHDL-basierte Implementation von Zufallszahlengeneratoren auf Chipkarten (SmartCards)*, Report FBI HH B 234 01, Universität Hamburg, Bibliothek des Fachbereichs Informatik, 2001
Available at **`http://tech-www.informatik.uni-hamburg.de/paper/2001/SA_Witt_Hartmann/SaWitt.Hartmann.ps.gz`**.

[Schn96] Bruce Schneier: *Applied Cryptography*, John Wiley & Sons, 1996

[3]The explanation from Sun is: “Sun’s JDK for Linux maps each java threads to a native threads in the OS. This works well on Solaris and Windows, but not so well on Linux because Linux (for better or worse) does not have a light weight thread model. As result on Linux when using the 1.3 (or latter) Sun JDK each thread a Java program spawns appears as a process in ps.”

[STW97] Dennis Shasha, Surendranath Talla, Peter Wyckoff: *An Approach to Fault-tolerant Parallel Processing on Intermittently Idle, Heterogeneous Workstations*, Proceedings of the 27th International Symposium on Fault-Tolerant Computing, 1997
Available at **`http://citeseer.nj.nec.com/38708.html`**.

[OW00] Scott Oaks, Henry Wong: *Jini In A Nutshell*, O'Reilly, 2000